ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
&
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ανάπτυξη εφαρμογής πιστοποιημένων μηνυμάτων με χρήση τεχνολογίας Blockchain

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

του

Γεράσιμου Βάρνη

(ΑΕΜ: 154)

Επιβλέπων : **Άγγελος Μιχάλας**

Ιδιότητα

Καστοριά, Απρίλιος - 2022

Η παρούσα σελίδα σκοπίμως παραμένει λευκή

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
&
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ανάπτυξη εφαρμογής πιστοποιημένων μηνυμάτων με χρήση τεχνολογίας Blockchain

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

του

Γεράσιμου Βάρνη

(ΑΕΜ: 154)

Επιβλέπων : **Άγγελος Μιχάλας**

Ιδιότητα

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την (εδώ συμπληρώνεται η ημερομηνία εξέτασης της εργασίας).

.....	.....	.....
Ον/μο Μέλους	Ον/μο Μέλους	Ον/μο Μέλους
Ιδιότητα Μέλους	Ιδιότητα Μέλους	Ιδιότητα Μέλους

Καστοριά, Απρίλιος - 2022

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέπων καθηγητή κ. Άγγελο Μιχάλα, για την ευκαιρία να ασχοληθώ με ένα εξαιρετικά ενδιαφέρον αντικείμενο έρευνας και φυσικά για την πολύτιμη βοήθεια του ως προς την περάτωση της παρούσας μεταπτυχιακής εργασίας.

Επίσης, σε αυτό το σημείο πρέπει να εκφράσω τον σεβασμό προς την οικογένεια μου και να τους ευχαριστήσω για την συνεχή και σημαντική συμπαράστασή τους καθ'όλη την διάρκεια των μεταπτυχιακών σπουδών.

Περίληψη

Σκοπός της παρούσας μεταπτυχιακής εργασίας είναι να αναπτυχθεί μία διαδικτυακή εφαρμογή ανταλλαγής μηνυμάτων μεταξύ 2 χρηστών προσδίδοντας έναν ικανοποιητικό βαθμό αξιοπιστίας και πιστοποίησης ως προς το περιεχόμενο του μηνύματος, που παραμένει αναλλοίωτο και προστατεύεται μέσω της τεχνολογίας Blockchain. Η τεχνολογία Blockchain αποτελεί ένα αντικείμενο μελέτης που απασχολεί ιδιαίτερα την ερευνητική κοινότητα τα τελευταία χρόνια και η διασημότερη εφαρμογή της είναι τα κρυπτονομίσματα, δεδομένου ότι ξεκίνησε σαν κρυπτογραφημένη βάση δεδομένων για την αποθήκευση των συναλλαγών bitcoin. Ωστόσο, οι εφαρμογές του blockchain πλέον έχουν εκτοξευθεί με τεράστιο ενδιαφέρον στην παγκόσμια τεχνολογική, προγραμματιστική κοινότητα και όχι μόνο.

Όπως θα δούμε στη συνέχεια, στην τρέχουσα μελέτη χρησιμοποιείται η τεχνολογία Blockchain μέσω του εργαλείου BigchainDB συνδυάζοντας παράλληλα και μία πιο παραδοσιακή βάση δεδομένων MongoDB / NoSQL. Έτσι, δημιουργείται ένα *υβριδικό μοντέλο* ανταλλαγής πιστοποιημένων μηνυμάτων, στο οποίο ο τοπικός υπολογιστής λειτουργεί σαν κόμβος του blockchain δικτύου BigchainDB και οι 2 βάσεις δεδομένων συνεργάζονται μεταξύ τους για να προκύψει το αποτέλεσμα της αμετάβλητης (immutable), αποκεντρωμένης (decentralized) και κατανεμημένης (distributed) εφαρμογής μηνυμάτων.

Λέξεις Κλειδιά: blockchain, κρυπτογραφία, προγραμματισμός, διαδίκτυο, βάση δεδομένων, εφαρμογή ιστοτόπου

Abstract

The purpose of this master thesis is to develop an online messaging application between 2 users providing a satisfactory degree of reliability and certification in terms of message content, which remains unchanged and is protected through Blockchain technology. Blockchain technology has been the subject of much study in the research community in recent years and its most popular application is cryptocurrencies, as it started as an encrypted database for storing bitcoin transactions. However, blockchain applications have now been launched with great interest in the global technological, programming community and beyond.

As we will see later, the current study uses Blockchain technology through the BigchainDB tool while combining a more traditional MongoDB / NoSQL database. Thus, a hybrid certified messaging model is created, in which the local computer acts as a node of the BigchainDB blockchain network and the two databases work together to obtain the result of the immutable, decentralized and distributed messaging application.

KeyWords: blockchain, cryptography, programming, internet, database, web application

Πίνακας Περιεχομένων

Λίστα Σχημάτων

1. Εισαγωγή

1.1 Πρόλογος

Διανύουμε μία εποχή στην οποία το διαδίκτυο αποτελεί μεγάλο μέρος της καθημερινότητας μας. Με την τεράστια τεχνολογική ανάπτυξη έχει επέλθει ταυτόχρονα και μεγάλη αύξηση στις πληροφορίες που μεταφέρονται, επομένως κατανοούμε τη σημασία της ασφαλούς αποθήκευσης, επεξεργασίας και ανταλλαγής των δεδομένων.

Έχουν αναπτυχθεί διάφορες τεχνικές για τη διασφάλιση της συνεχούς και αδιάλειπτης εξυπηρέτησης των χρηστών εφαρμογών ιστοτόπου, όπως δημιουργία αντιγράφων ασφαλείας ή ενημέρωση δεδομένων σε πραγματικό χρόνο. Οι περισσότερες διαδικτυακές εφαρμογές μικρομεσαίας κλίμακας χρησιμοποιούν κυρίως παραδοσιακές βάσεις δεδομένων (Sql, NoSql) για την αποθήκευση πληροφοριών, την ανάκτηση και επεξεργασία τους ανά πάσα στιγμή. Ωστόσο, ιδιαίτερα τα τελευταία χρόνια με την ραγδαία άνοδο των Big Data και την ταυτόχρονη αύξηση κυβερνοεπιθέσεων σε πληροφοριακά συστήματα, υπάρχει έντονη ανάγκη για υψηλή αξιοπιστία σε όλες τις ενέργειες που σχετίζονται με τα δεδομένα (π.χ. μεταφορά, καταχώρηση, τροποποίηση) με συνεχή έρευνα για όλο και ασφαλέστερες μεθόδους. Σε αυτό το φάσμα κινείται η τεχνολογία blockchain, που συνοπτικά είναι μία πρωτοποριακή βάση δεδομένων με αποκεντρωμένο, κατανεμημένο και αμετάβλητο χαρακτήρα.

1.2 Δομή Εργασίας

Η παρούσα εργασία διαχωρίζεται σε 3 σημαντικά κεφάλαια. Αρχικά, στο 2ο κεφάλαιο παρουσιάζονται όλα τα στοιχεία που χαρακτηρίζουν την τεχνολογία blockchain, ήτοι πως ξεκίνησε και πως εξελίχθηκε ως τώρα, ποιά είναι η δομή της και πως λειτουργεί και που έχει εφαρμοστεί μέχρι στιγμής. Στο 3ο κεφάλαιο αναφέρονται λεπτομερώς όλες οι τεχνολογίες και τα εργαλεία που χρησιμοποιήθηκαν στην ανάπτυξη της εφαρμογής. Τέλος, στο 4ο κεφάλαιο θα πραγματοποιηθεί παρουσίαση της λειτουργίας της εφαρμογής με παράθεση κώδικα και πρακτική επεξήγηση.

2. Blockchain

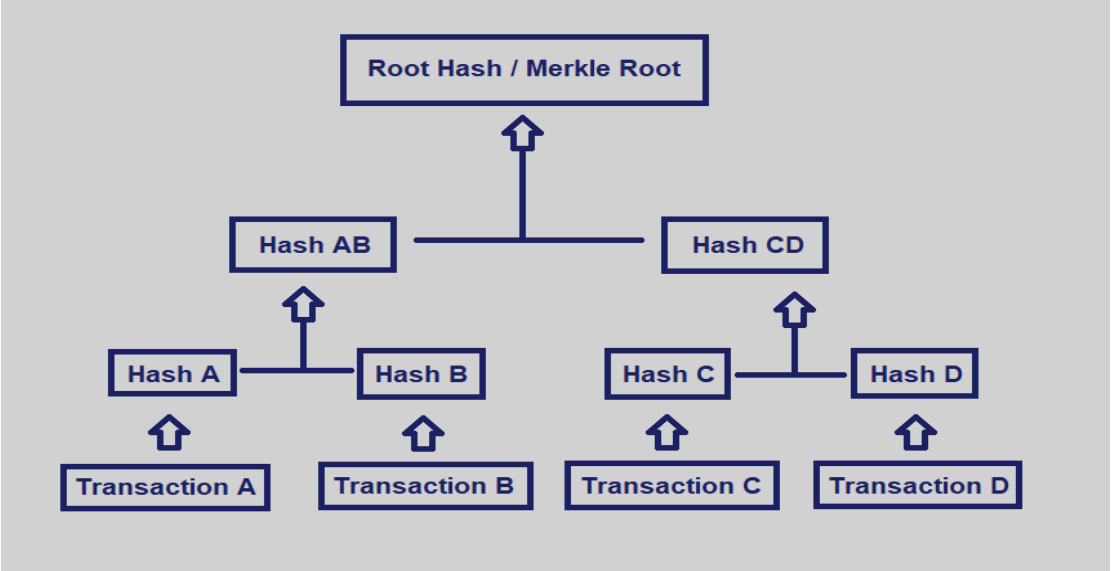
2.1 Ιστορική Αναδρομή

Για να φτάσουμε στην υλοποίηση του πρώτου ολοκληρωμένου συστήματος blockchain το 2009, προηγήθηκαν διάφορα σημαντικά γεγονότα που οδήγησαν στην πραγμάτωση αυτής της πρωτοποριακής ιδέας.

Αρχικά, το 1982 ο David Chaum μελέτησε την τεχνολογία blockchain σε πρώιμη μορφή προσπαθώντας να περιγράψει την ασφαλή επικοινωνία διαφόρων άγνωστων μερών μέσα σε ένα υπολογιστικό σύστημα και να εξάγει κάποιο κρυπτογραφικό μοντέλο, που θα μπορούσε να χρησιμοποιηθεί ως αξιόπιστος τρόπος ανταλλαγής και αποθήκευσης δεδομένων για όλους τους χρήστες. Σε παρόμοιο μοτίβο κινήθηκαν και οι Haber-Stornetta το 1991, που προσπάθησαν να δημιουργήσουν ένα ψηφιακό σύνολο από blocks στα οποία θα αποθηκεύονταν διάφορα αρχεία και δεν θα ήταν εφικτό να αλλοιωθεί ο χρόνος που έγινε η εισαγωγή του κάθε αρχείου στο σύστημα. Η ιδέα της αλυσίδας από κουτιά/blocks (blockchain) ήρθε πιο κοντά στην σημερινή μορφή της, όταν το 1992 ο Bayer εφάρμοσε σε συνεργασία με τους Haber-Stornetta τον κατακερματισμό* πάνω στο μοντέλο του 1991 με ερεθίσματα από τα Merkle trees (*Εικόνα 1* / μία δενδροειδής δομή δεδομένων στην οποία κάτω κάτω υπάρχουν οι συναλλαγές με τα δεδομένα τους και όσο ανεβαίνουμε επίπεδο γίνεται συνεχώς κρυπτογράφηση των δεδομένων του προηγούμενου επιπέδου μέχρι να φτάσουμε σε ένα μόνο hash), με αποτέλεσμα να είναι όλες οι συναλλαγές συνδεδεμένες μεταξύ τους και κατ'επέκταση να είναι αρκετά δύσκολο να τροποποιηθεί το περιεχόμενο της αλυσίδας.

* *Ο κατακερματισμός είναι ένας αλγόριθμος κρυπτογραφίας που λαμβάνει κάποια δεδομένα ως είσοδο/input μορφής και ως αποτέλεσμα προκύπτει μία έξοδος/output κρυπτογραφημένης μορφής, γνωστό και ως hash. Το σημαντικό χαρακτηριστικό του κατακερματισμού είναι πως όταν εισάγονται τα ίδια δεδομένα θα προκύπτει πάντα το ίδιο αποτέλεσμα και έτσι ελέγχονται τιμές hash, καθώς είναι πολύ δύσκολο να προκύψει η είσοδος αν είναι γνωστή η έξοδος. Περισσότερες λεπτομέρειες για τον αλγόριθμο hash που χρησιμοποιείται στην τρέχουσα εργασία θα δοθούν στο κεφάλαιο 3, ωστόσο κρίθηκε αναγκαίο να ενημερωθεί ο αναγνώστης γενικά για την έννοια του hash που αναφέρεται αρκετά στη συνέχεια.*

Περίπου 16 χρόνια αργότερα, το 2008 δημιουργήθηκε το πρώτο κρυπτονόμισμα υπό την ονομασία bitcoin με κυκλοφορία του επίσημου ιστοτόπου *bitcoin.org* και φημολογείται ότι αυτό έγινε από κάποιο μεμονωμένο άτομο ή ένα σύνολο ατόμων με το προσωνύμιο Satoshi Nakamoto. Η αναλυτική παρουσίαση της λογικής και της φιλοσοφίας σχετικά με το κρυπτονόμισμα δημοσιεύτηκε από την ίδια ομάδα ανθρώπων σε μία αναφορά με τίτλο *'Bitcoin: A Peer-to-Peer Electronic Cash System'* , όπου και γίνεται μία σύντομη εισαγωγή στην τεχνολογία blockchain πάνω στην οποία βασίζεται το σύστημα bitcoin. Μέχρι το 2014 οι εφαρμογές περιορίζονταν γενικά στα κρυπτονομίσματα και κυρίως στο bitcoin, ωστόσο η μεγάλη στροφή στην πορεία του blockchain έγινε με την κυκλοφορία του Ethereum το 2015. Επρόκειτο για μία νέα μορφή blockchain που υλοποιήθηκε από προγραμματιστές (ανάμεσα τους και αρκετοί άνθρωποι που είχαν βοηθήσει στην ανάπτυξη του bitcoin), οι οποίοι κατάφεραν να 'ξεκλειδώσουν' και εντέλει να προσφέρουν καινούριες δυνατότητες, πρωτότυπους τρόπους χρήσης του blockchain πέραν των συστημάτων κρυπτονομίσματος. Σταδιακά από τότε και μέχρι το τρέχον έτος 2022, άρχισε μία σειρά πειραμάτων από εταιρείες σε παγκόσμιο επίπεδο για ενσωμάτωση και εκμετάλλευση του blockchain σε διάφορους τομείς π.χ. χρήση σε ενδοεταιρικές εφαρμογές ανταλλαγής αρχείων που αφορούν σημαντικά δεδομένα.

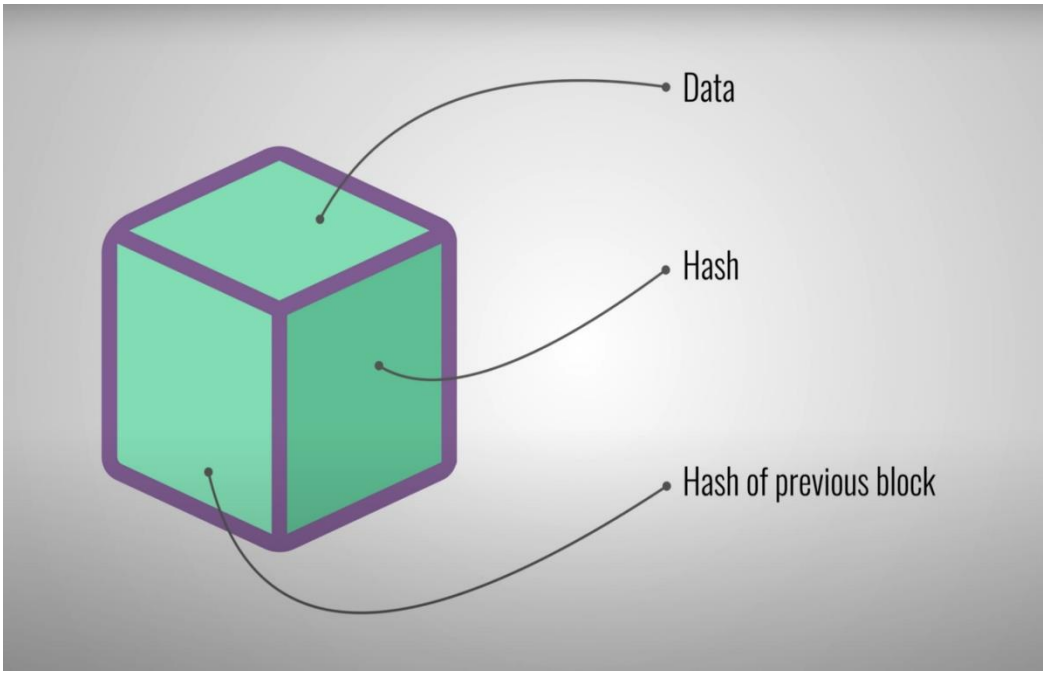


Εικόνα 1 - Παράδειγμα Merkle Tree

2.2 Γενικότερα περί Blockchain

Προτού προχωρήσουμε στην ανάλυση της δομής και στα χαρακτηριστικά του blockchain, κρίνεται σκόπιμο να αναφερθεί σύντομα ένας ορισμός της τεχνολογίας και να έχει μία κάποια γενική εικόνα ο αναγνώστης.

Ουσιαστικά, το blockchain μπορεί να θεωρηθεί ως ένα διαφορετικό είδος βάσης δεδομένων στην οποία οι πληροφορίες τοποθετούνται με ψηφιακή μορφή σε διάφορα κουτιά (blocks) που είναι συνδεδεμένα μεταξύ τους σαν αλυσίδα (chain) με την χρήση κρυπτογραφίας, ήτοι κάθε block περιέχει α) τα δεδομένα του, β) ένα hash από τα δεδομένα του και γ) ένα hash από τα δεδομένα του προηγούμενου block (*Εικόνα 2*). Επιπλέον όπως θα δούμε, ένα σύστημα που χρησιμοποιεί blockchain χαρακτηρίζεται γενικά από υψηλό επίπεδο ασφάλειας στις ανταλλαγές δεδομένων και εύκολο εντοπισμό τυχόν τροποποιήσεων σε αυτά, προσφέροντας τις περισσότερες φορές αυτόματη επαναφορά της φυσιολογικής κατάστασης της αλυσίδας.



Εικόνα 2 - Περιεχόμενο ενός block

Σε αυτό το σημείο, προκειμένου να μην δημιουργηθούν ασάφειες και ερωτηματικά σχετικά με την διαφορά ενός κρυπτονομίσματος από την τεχνολογία blockchain, πρέπει να αναφερθούν τα παρακάτω:

- Ένα κρυπτονόμισμα είναι απλώς μία μορφή ηλεκτρονικού νομίσματος, που χρησιμοποιείται για να γίνει μία ανταλλαγή δεδομένων μεμονωμένα μεταξύ 2 χρηστών (χωρίς να εμπλέκεται κάποιος άλλος) και αποθηκεύεται στο δίκτυο του blockchain με την χρήση κρυπτογραφίας, ενώ ο ορισμός για αυτή την ενέργεια ανταλλαγής είναι συναλλαγή/transaction. Παραδείγματα τέτοιων νομισμάτων αποτελούν το bitcoin, ethereum, tether, dogecoin και πολλά άλλα.
- Το blockchain αποτελείται από κουτιά/blocks, τα οποία περιέχουν το καθένα μία και παραπάνω συναλλαγές και τις πληροφορίες που αντιστοιχούν στην κάθε συναλλαγή. Παρόλα αυτά, για να γίνει μία συναλλαγή σε κάποιο blockchain μπορεί να μην χρειάζεται η καταβολή ενός ποσού κρυπτονομίσματος ή οι πληροφορίες που συνοδεύουν το κάθε transaction να μην περιέχουν κρυπτονόμισμα, αλλά κάτι διαφορετικό όπως για παράδειγμα κάποιο κείμενο.
- Συμπερασματικά, τα κρυπτονομίσματα χρησιμοποιήθηκαν ως η πρώτη εφαρμογή της τεχνολογίας blockchain στοχεύοντας στο να μπορούν να γίνονται με ασφάλεια και αξιοπιστία συναλλαγές μεταξύ ατόμων που δεν γνωρίζονται, χωρίς την ανάγκη για μεσολάβηση άλλων ατόμων.
- Άρα, το blockchain είναι μία γενικότερη έννοια σε σχέση με το κρυπτονόμισμα και αποτελεί όπως είπαμε μία πρωτοποριακή βάση δεδομένων στην οποία αποθηκεύονται οι συναλλαγές μεταξύ χρηστών είτε γίνονται με την χρήση κρυπτονομισμάτων είτε όχι, προσφέροντας τρομερά πλεονεκτήματα.

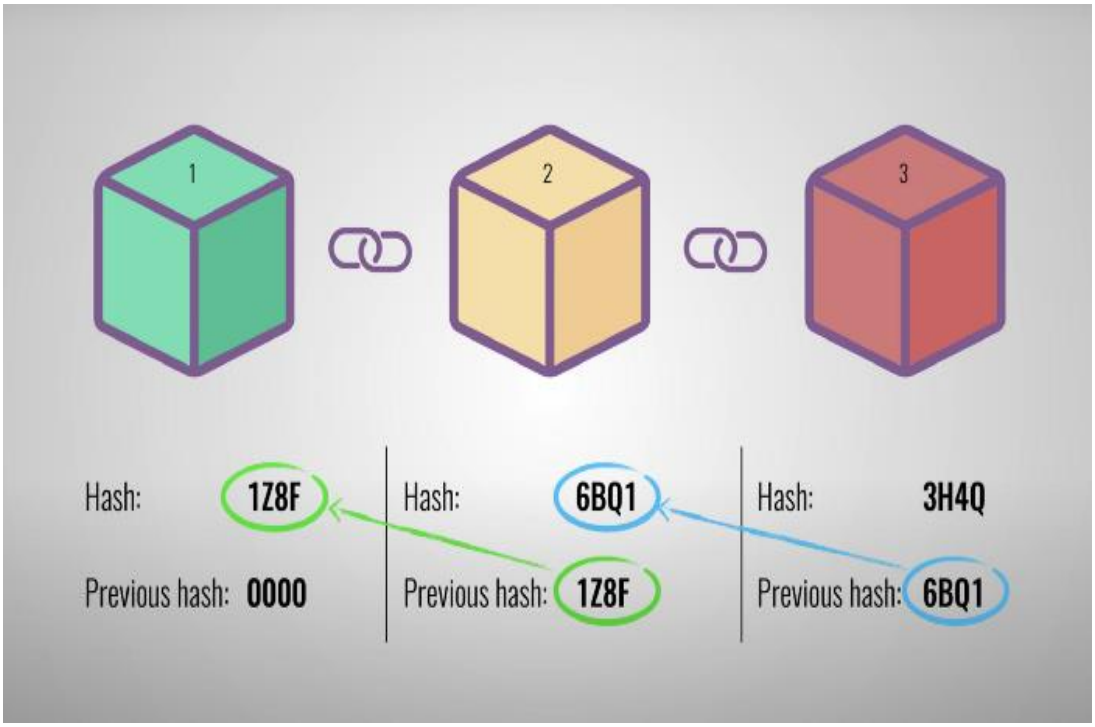
2.3 Αρχιτεκτονική της τεχνολογίας Blockchain

2.3.1 Βασικά χαρακτηριστικά

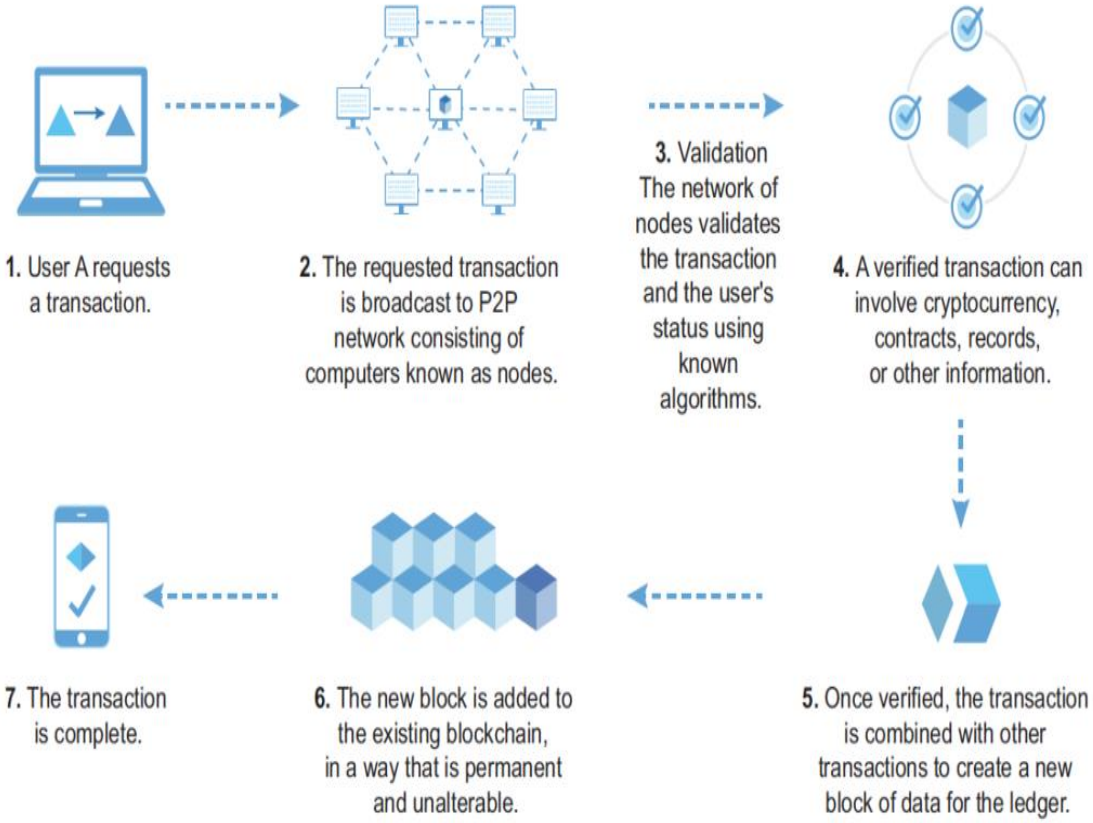
Ξεκινώντας την λεπτομερή ανάλυση για το blockchain, πρέπει να δοθεί μία πιο κατατοπιστική περιγραφή και να τονιστούν τα 3 σημαντικά στοιχεία του. Στη γενική του μορφή πρόκειται για μία βάση δεδομένων με 1) αποκεντρωμένο και 2) κατανεμημένο χαρακτήρα καθώς τα δεδομένα δεν αποθηκεύονται σε ένα μόνο υπολογιστικό σύστημα ούτε διαχειρίζονται από μία μεμονωμένη οντότητα (ωστόσο έχουν αναπτυχθεί ιδιωτικά blockchain που μειώνουν τον βαθμό αποκέντρωσης), αλλά διαμοιράζονται σε όλους τους κόμβους του δικτύου (κάθε κόμβος έχει το ίδιο αντίγραφο του blockchain) οι οποίοι συγκεντρωτικά τα ελέγχουν ακολουθώντας ένα μοντέλο συναίνεσης. Επίσης, τα δεδομένα ενός blockchain δεν μπορούν να τροποποιηθούν καθώς αυτό προυποθέτει να γίνει ταυτόχρονα η ίδια αλλαγή σε όλους τους κόμβους του δικτύου blockchain (κάτι που πρακτικά είναι ανέφικτο διότι οι υπολογιστές είναι διαμοιρασμένοι σε όλο τον κόσμο, *τουλάχιστον στα δημόσια blockchain*), άρα το blockchain είναι 3) αμετάβλητο.

Στην παρακάτω εικόνα φαίνεται μία απλή μορφή blockchain με 3 blocks και παρατίθεται για να χρησιμοποιηθεί ως παράδειγμα επεξήγησης της δομής του.

- Το block 1 ονομάζεται genesis block και είναι το πρώτο κουτί του blockchain και το μοναδικό που τοποθετείται χειροκίνητα/hardcoded από τους προγραμματιστές, όταν δημιουργείται το blockchain. Το previous hash είναι 0000 ή μπορεί να είναι οτιδήποτε άλλο όπως πχ ένα string *'this is the genesis block'*, καθώς δεν υπάρχει προηγούμενο block στο οποίο να αναφέρεται αυτό το previous hash. Το hash είναι η κρυπτογραφημένη μορφή των δεδομένων του genesis block.
- Το κουτί/block 2 συνδέεται με το προηγούμενο κουτί, εφόσον το previous hash του 1*Z8F* είναι το hash του genesis block. Παρομοίως, το κουτί 3 συνδέεται με το κουτί 2 μέσω του previous hash *6BQ1*.
- Άρα, όποιο block προστίθεται θα συνδέεται και με το προηγούμενο και με το επόμενο κουτί λόγω του previous hash. Έτσι, μεγαλώνει συνεχώς η αλυσίδα έχοντας 'ανοσία' στις αλλαγές των δεδομένων.



Εικόνα 3 - Αναπαράσταση ενός blockchain, που αποτελείται από 3 blocks



Εικόνα 4 - Επεξήγηση λειτουργίας blockchain

2.3.2 Λειτουργία Blockchain

Προκειμένου να λειτουργήσει μία εφαρμογή με χρήση blockchain χρειάζεται να βασίζεται πάνω σε ένα δίκτυο peer-to-peer ή αλλιώς P2P. Στην επικρατέστερη περίπτωση του δημοσίου blockchain σημαίνει, πως ο κάθε κόμβος του δικτύου P2P έχει το ίδιο αντίγραφο της αλυσίδας και τα ίδια δικαιώματα χρήσης του, δηλαδή είναι ισότιμος με όλους τους υπόλοιπους κόμβους ως προς την πρόσβαση και τις επιτρεπτές ενέργειες πάνω στο blockchain. Επιπλέον, στην κλασσική έκδοση του blockchain δεν υπάρχει κάποιος κεντρικός υπολογιστής που θα λειτουργεί ως server και θα μεσολαβεί στην επικοινωνία των κόμβων, αλλά η επικοινωνία τους γίνεται απευθείας και όλα τα δεδομένα περνάνε από όλους τους κόμβους, αφού η κάθε συναλλαγή επικυρώνεται καθολικά (πλέον έχουν δημιουργηθεί και μοντέλα που ενσωματώνουν κόμβους διαχειριστές με περισσότερα δικαιώματα, αλλά αυτό γίνεται κυρίως στα πλαίσια εταιρειών που δημιουργούν το δικό τους ιδιωτικό blockchain).

Συνοπτικά και με βάση την *Εικόνα 4,* ένα δίκτυο blockchain λειτουργεί ως εξής:

- Ένας χρήστης/αποστολέας δημιουργεί μία συναλλαγή και την κρυπτογραφεί με το δημόσιο κλειδί του παραλήπτη και το ιδιωτικό του κλειδί. Επίσης, την υπογράφει πάλι με το ιδιωτικό του κλειδί. (βήμα 1)
- Η συναλλαγή στέλνεται σε όλους τους κόμβους και μέσω της ψηφιακής υπογραφής επικυρώνεται. (βήμα 2 & 3)
- Αν πρόκειται για έγκυρη συναλλαγή, τότε τοποθετείται είτε μόνη της είτε μαζί με άλλες εκκρεμείς συναλλαγές σε κάποιο προσωρινό κουτί. (βήμα 4 & 5)
- Διαδικασία συναίνεσης με την οποία προστίθεται το κουτί στην αλυσίδα. (μετά το βήμα 5 και στο βήμα 6)
- Ο χρήστης/παραλήπτης λαμβάνει την συναλλαγή και την αποκρυπτογραφεί μέσω του δικού του ιδιωτικού κλειδιού. (βήμα 7)

2.3.2.1 Public & Private Keys

Παρόλα αυτά, υπάρχει κάτι που διαφοροποιεί τον κάθε υπολογιστή του δικτύου από τους άλλους και πρόκειται για το μοναδικό ζευγάρι δημόσιου και ιδιωτικού κλειδιού (public & private key), έννοιες που συνδέονται με την κρυπτογράφηση. Γενικά, ο αποστολέας χρησιμοποιεί το ιδιωτικό του κλειδί και

το δημόσιο κλειδί του παραλήπτη για να κρυπτογραφήσει μία επιθυμητή ανταλλαγή δεδομένων και ο παραλήπτης όταν λάβει τα δεδομένα, τα αποκρυπτογραφεί κάνοντας χρήση του δικού του ιδιωτικού κλειδιού και του δημοσίου του αποστολέα.

Συγκεκριμένα όμως, στο blockchain τα κλειδιά χρησιμοποιούνται α) για να κρυπτογραφηθούν οι συναλλαγές με το δημόσιο κλειδί του παραλήπτη , β) να γίνει η ψηφιακή υπογραφή της συναλλαγής με το ιδιωτικό κλειδί του αποστολέα, όπως επίσης και γ) για την επικύρωση της υπογραφής με το δημόσιο κλειδί. Το δημόσιο κλειδί του κάθε χρήστη/κόμβου είναι ορατό σε όλους και χρησιμεύει ως μία διεύθυνση στην οποία πρόκειται να σταλούν συναλλαγές, ενώ το ιδιωτικό κλειδί είναι γνωστό μόνο σε αυτόν που ανήκει. Έτσι, αν κάποιος χρήστης θέλει να δημιουργήσει μία συναλλαγή προς κάποιον άλλο χρήστη, αρχικά κρυπτογραφεί τις πληροφορίες με το δημόσιο κλειδί του παραλήπτη και το δικό του ιδιωτικό κλειδί. Έπειτα, υπογράφει ψηφιακά την συναλλαγή με το δικό του ιδιωτικό κλειδί (ιδιωτικό κλειδί αποστολέα) και η συναλλαγή στέλνεται σταδιακά σε όλους τους κόμβους για έλεγχο εγκυρότητας (*βήμα 2/Εικόνα 4*). Σύμφωνα με αυτοματοποιημένους αλγορίθμους επιβεβαίωσης, πρακτικά προκύπτει από την ψηφιακή υπογραφή της κάθε συναλλαγής μία μοναδική διεύθυνση ή αλλιώς το μοναδικό δημόσιο κλειδί και συγκρίνεται αν ισούται με το δημόσιο κλειδί του ατόμου, που ισχυρίζεται ότι έστειλε την συναλλαγή (*βήμα 3/Εικόνα 4*). Αν η διαδικασία δώσει έγκυρο αποτέλεσμα τότε η συναλλαγή πρόκειται να περιέχεται στο επόμενο κουτί/block που θα προστεθεί στο blockchain μαζί με άλλα transactions, κάτι το οποίο γίνεται με συγκεκριμένο τρόπο που έχει οριστεί από το εκάστοτε μοντέλο - αλγόριθμο συναίνεσης.

2.3.2.2 Συναλλαγή - Transaction

Με απλά λόγια, συναλλαγή σε ένα δίκτυο blockchain ονομάζεται η ανταλλαγή δεδομένων μεταξύ 2 χρηστών και αυτά τα δεδομένα μπορεί να είναι ως γνωστόν κρυπτονομίσματα που μεταφέρονται από ένα ηλεκτρονικό πορτοφόλι σε κάποιο άλλο, όμως μπορεί να έχουν και τελείως διαφορετική μορφή για παράδειγμα μηνύματα κειμένου (όπως η τρέχουσα εφαρμογή μηνυμάτων) ή ηλεκτρονικά έγγραφα, αρχεία μουσικής, αρχεία εικόνας (αυτά είναι ακόμα σε ερευνητικό στάδιο).

Συνήθως ένα transaction σε κάποιο σύστημα που περιλαμβάνει blockchain είναι ένα αντικείμενο (object γλώσσας προγραμματισμού), το οποίο περιέχει κάποιες ιδιότητες. Αυτές οι ιδιότητες είναι αυτό που καλείται περιεχόμενο συναλλαγής και οι κυριότερες είναι το μοναδικό Id της συναλλαγής, το Timestamp που είναι το πότε δημιουργήθηκε η συναλλαγή, το SenderPublicKey και RecipientPublicKey ήτοι τα δημόσια κλειδιά του αποστολέα και παραλήπτη, το Data που πρακτικά είναι τα δεδομένα και μπορούν να περιέχουν ποσό κρυπτονομίσματος ή μήνυμα κειμένου (κρυπτογραφημένο και όχι ως απλό string, όπως φαίνεται εδώ για λόγους επεξήγησης) ή επιμέρους αντικείμενο με πληροφορίες και η ψηφιακή υπογραφή DigitalSignature που χρησιμοποιείται από τους κόμβους για έλεγχο εγκυρότητας. Δηλαδή η συναλλαγή θα είναι της μορφής (τυχαίες τιμές):

transaction = {

"Id": "135486913",

"Timestamp": "16:15:44 02/11/2021",

"DigitalSignature":"cad8bfa461829ec59c2dda78cc9042e30976574deaeda27c306e4f1
0f6770d0bbd470d7f2e3bf1a962247da88529391d"

"SenderPublicKey":"990552c9675aca8b04db925234fcc441ecc5c172a98e29f1ca4cc32
61c22cd60b2a06d45c2513a3174a65a921b23fe4b",

"RecipientPublicKey":"74897963d85c8500c6fbc22276cb4c7b12e4bd9180818b5e82c
a5d943cae9dd85c7212fe3dfbf4fe06c513c944a24a41",

"Data": "encrypted text message or another object",

}

Έτσι, αφού επικυρωθούν οι συναλλαγές τοποθετούνται σε κάποιο προσωρινό block (*βήμα 5/Εικόνας 4*) και ακολουθεί η διαδικασία συναίνεσης για την προσθήκη του στο blockchain.

2.3.2.3 Μοντέλα Συναίνεσης

Λοιπόν, έχουν ήδη γίνει αρκετές αναφορές στον όρο συναίνεση, αλγόριθμος συναίνεσης, μοντέλο συναίνεσης σε ένα δίκτυο P2P που λειτουργεί με blockchain και ουσιαστικά όλες οι έννοιες αντικατοπτρίζουν τον τρόπο, την μέθοδο που ακολουθείται ώστε να ληφθεί η απόφαση για την προσθήκη ενός block ή όχι στην αλυσίδα (μετά το *βήμα 5/Εικόνα 4* και πριν το *βήμα 6/Εικόνα 4*, το κουτί που περιέχει όλες τις εκκρεμείς συναλλαγές τοποθετείται σε ένα προσωρινό σημείο

μέχρι να βγει το αποτέλεσμα του αλγορίθμου συναίνεσης. Επομένως, η συναίνεση συμβαίνει ανάμεσα στο βήμα 5 και βήμα 6 της προηγούμενης εικόνας). Σε αυτή την διαδικασία ανάλογα με το μοντέλο θα υπάρχουν φυσικά διάφορες προυποθέσεις που το χαρακτηρίζουν και σκοπός της είναι να προκύπτει κάθε φορά μόνο μία μελλοντική κατάσταση του blockchain, κοινώς αποδεκτή από όλους τους χρήστες χωρίς να τίθεται θέμα αξιοπιστίας.

Πλέον, έχουν αναπτυχθεί δεκάδες πρωτόκολλα συναίνεσης με τα σημαντικότερα και πιο διαδεδομένα να είναι τα εξής:

- Proof of Work / PoW: Πρόκειται για το διασημότερο μοντέλο που χρησιμοποιείται στο σύστημα bitcoin (αλλά και σε άλλα κρυπτονομίσματα) και περιλαμβάνει τις πολύ γνωστές έννοιες mining, difficulty, nonce. Ο τρέχων αλγόριθμος για να προσδώσει το επιθυμητό αποτέλεσμα της προσθήκης καινούριου block στην αλυσίδα, περιλαμβάνει την εύρεση λύσης σε δύσκολες μαθηματικές πράξεις κατακερματισμού-hash και η διαδικασία αυτή ονομάζεται mining. Πιο συγκεκριμένα, στο δίκτυο του bitcoin ρυθμίζεται αυτόματα μία τιμή difficulty ώστε ο μέσος χρόνος προσθήκης block να είναι περίπου 10 λεπτά και υποδηλώνει πόσο δύσκολο είναι να βρεθεί η λύση του προβλήματος, δηλαδή να βρεθεί μία τιμή hash 64 ψηφίων δεκαεξαδικής μορφής που αριθμητικά είναι μικρότερη η ίση με ένα δεδομένο hash (target hash).

  Ο τρόπος που προσπαθεί ο κάθε χρήστης να βρεί την τιμή hash και να ολοκληρώσει το mining, είναι με αλλεπάλληλες δοκιμές πάνω στο hashed block header (το οποίο είναι μοναδικό και αντιστοιχεί μόνο στο block, το εκάστοτε block που πρόκειται να προστεθεί στο blockchain και περιέχει πληροφορίες για τις συναλλαγές του block) αλλάζοντας κάθε φορά μόνο ένα νούμερο που ονομάζεται nonce. Ο γρηγορότερος miner που θα μπορέσει να δημιουργήσει το hash που αντιστοιχεί αριθμητικά σε μικρότερη ή ίση τιμή με το επιθυμητό block header (από το οποίο προκύπτει το target hash), παίρνει ως ανταμοιβή ένα ποσό bitcoin και το κουτί προστίθεται στην αλυσίδα. Για να γίνουν αυτές οι πολύπλοκες πράξεις κατακερματισμού και να μπορέσουν να κερδίσουν

οι miners, σημαίνει ότι έχουν δουλέψει για το αποτέλεσμα (αυτή είναι η απόδειξη της δουλειάς, το proof of work) και υποχρεωτικά διαθέτουν υπολογιστικά συστήματα με απίστευτα μεγάλες επεξεργαστικές ικανότητες, αλλιώς θα ήταν ανέφικτο να *τρέξουν* πρώτοι τους αλγόριθμους hash και να γίνει το επιτυχημένο mining.

- Proof of Elapsed Time / PoET: Υλοποιήθηκε από την πασίγνωστη εταιρεία Intel και βασίζεται σε μία γεννήτρια παραγωγής τυχαίων αριθμών που αντιστοιχούν σε χρόνους αναμονής. Δημιουργούνται λοιπόν τυχαίοι χρόνοι που πρέπει να περιμένει ο κάθε χρήστης του δικτύου blockchain και αυτός που περίμενε λιγότερο, είναι αυτός που κερδίζει το δικαίωμα να προσθέσει το νέο κουτί στο blockchain. Ωστόσο, σύμφωνα με τις προδιαγραφές του μοντέλου proof of elapsed time είναι υποχρεωτικό να ελέγχονται και να καταγράφονται οι χρόνοι αναμονής όλων των κόμβων και η συνολική δικτυακή συμπεριφορά τους, ώστε να διασφαλίζεται ότι το αποτέλεσμα ήταν όντως τυχαίο και αν η διαδικασία διενεργήθηκε με διαφάνεια, τότε πραγματοποιείται η ενημέρωση του blockchain με το καινούριο block σε όλο το δίκτυο.
- Proof of Stake / PoS: Αποτελεί ίσως τον 2ο διασημότερο αλγόριθμο συναίνεσης σε ένα δίκτυο blockchain, μετά το proof of work και υπάρχει στον αλγόριθμο tendermint που χρησιμοποιείται στο BigchainDB, όπως θα δούμε στη συνέχεια. Στο proof of stake όλοι οι χρήστες μπορούν να ποντάρουν κάποιο ποσό κρυπτονομίσματος, προκειμένου να αποκτήσουν δικαίωμα συμμετοχής στη διαδικασία που τελικά θα οδηγήσει στην προσθήκη νέου block στην αλυσίδα. Έτσι, ο κάθε χρήστης προτείνει κάποιο block που θεωρεί ότι μπορεί να γίνει το επόμενο στο blockchain και στέλνει το ποσό που επιθυμεί ως stake. Το ποσό που έχει δεσμεύσει προσωρινά ο χρήστης, του δίνει την πιθανότητα να επιλεγεί ως επικυρωτής/validator, δηλαδή να είναι το δικό του προτεινόμενο block που θα επεκτείνει την αλυσίδα (εφόσον θεωρηθεί ότι πληρεί όλες τις απαιτήσεις ενός νέου block π.χ. ότι έχει previous hash τιμή, που ισούται με το hash του προηγούμενου block). Αναφερθήκαμε στην πιθανότητα επιλογής ενός κόμβου ως validator

του επόμενου block, καθώς αυτό δεν εξαρτάται μόνο από το ποσό κρυπτονομίσματος που ποντάρει κάποιος αλλά υπάρχουν διάφορες επιπρόσθετες μεταβλητές που κάθε φορά θα παίζουν ρόλο στον αλγόριθμο.

Συνήθως, είναι η παλαιότητα ενός κόμβου ή το λεγόμενο coin age και ουσιαστικά δείχνει πόσο καιρό προσπαθεί ένας κόμβος να γίνει επικυρωτής, δηλαδή πόσο καιρό το ποσό που είχε ποντάρει έχει παραμείνει δεσμευμένο και μεγαλύτερη τιμή σημαίνει μεγαλύτερη πιθανότητα να επιλεγεί στο άμεσο μέλλον. Επίσης, η δεύτερη παράμετρος που δύναται να συμπεριλαμβάνεται στο μοντέλο proof of stake είναι η δημιουργία ενός hash string για όλους τους κόμβους ως αποτέλεσμα μίας τυχαίας ακολουθίας συμβόλων και συνδυαστικά με το ποντάρισμα τους προκύπτει για τον καθένα μία τιμή. Το καλύτερο αποτέλεσμα, η μεγαλύτερη τιμή κερδίζει και πρακτικά εμποδίζεται αυτό που συμβαίνει στο proof of burn (να ευνοείται συνεχώς αυτός που απλά ποντάρει τα περισσότερα κρυπτονομίσματα), καθώς η τιμή που βγαίνει από τον αλγόριθμο εμπεριέχει και τον παράγοντα τύχη. Εφόσον κάποιος χρήστης γίνει validator και προτείνει ένα block, μετά ακολουθεί η έγκριση του block από όλους τους υπόλοιπους κόμβους και τότε προστίθεται στην αλυσίδα, με το ποσό πονταρίσματος να επιστρέφει στον χρήστη (υπάρχουν μερικές παραλλαγές του παρόντος μοντέλου που επιβραβεύουν τον επικυρωτή επιπρόσθετα με κάποιο ποσό κρυπτονομίσματος, αλλά στην γενική μορφή δεν δίνεται ανταμοιβή). Σε διαφορετική περίπτωση μη έγκρισης του block, ο χρήστης χάνει το stake και η διαδικασία ξεκινά από την αρχή με την εκλογή νέου validator κόμβου.

- Proof of Activity / PoA: Μία ενδιαφέρουσα συνύπαρξη των σημαντικότερων μοντέλων proof of stake και proof of work, είναι το proof of activity και διαιρείται σε 2 στάδια. Αρχικά, περιλαμβάνει το mining του μελλοντικά νέου κουτιού στο blockchain με βάση την επεξεργαστική ισχύ των κόμβων, σύμφωνα με το proof of work και έπειτα η λειτουργία του μοντέλου μεταβαίνει στο proof of stake. Σε

αυτό το σημείο έχει ήδη βρεθεί το κουτί που πρόκειται να προστεθεί στην αλυσίδα, αλλά δεν έχει επικυρωθεί. Επομένως, με βάση την διαδικασία που αναφέρθηκε προηγουμένως γίνονται πονταρίσματα από όλους τους πιθανούς validators και επιλέγονται κάποιοι ή κάποιος για να το επικυρώσουν. Η επιλογή των validators (συνήθως είναι παραπάνω από ένας) πραγματοποιείται εν μέρει τυχαία, καθώς πάλι λαμβάνεται υπόψην το ποσό πονταρίσματος αλλά δεν αποτελεί τον μοναδικό παράγοντα επιλογής. Ωστόσο, εδώ δεν προτείνεται κάποιο κουτί από τους επικυρωτές και η δουλειά τους είναι απλά να ελέγξουν αν το κουτί από το mining είναι στη σωστή μορφή για να προστεθεί στην αλυσίδα.

Οι απόψεις διίστανται για το συγκεκριμένο πρωτόκολλο συναίνεσης, από τη μία μεριά πολλοί θεωρούν ότι δεν είναι ιδιαίτερα χρήσιμο αφού πάλι χρειάζονται πολύ δυνατά υπολογιστικά συστήματα και κατ'επέκταση μεγάλη κατανάλωση ηλεκτρικού ρεύματος για να λειτουργήσει, αλλά από την άλλη μεριά επικρατεί η άποψη ότι ενισχύει την αποκέντρωση του συστήματος blockchain, γιατί η ευθύνη προσθήκης νέου κουτιού διαμοιράζεται μεταξύ miners και validators (λόγω του συνδυαστικού αλγορίθμου, είναι εξαιρετικά σπάνιο να προκύπτει συνεχώς ο ίδιος miner που να είναι και validator και πολλές φορές αυτό ελέγχεται). Σε κάθε περίπτωση ο miner και οι επικυρωτές λαμβάνουν ανταμοιβή, όταν το κουτί προστεθεί στην αλυσίδα.

- Proof of Capacity / PoC: Σε αντίθεση με την περίπτωση του proof of work, στο proof of capacity οι χρήστες που συμμετέχουν στο blockchain δεσμεύουν χώρο στον σκληρό δίσκο τους προκειμένου να μπορέσουν να κάνουν mine το επόμενο block. Συγκεκριμένα, αντίστοιχα με το proof of work πάλι ο στόχος είναι να βρεθεί μία τιμή hash με βάση το block header του νέου κουτιού που θα προστεθεί, όμως εδώ δεν γίνονται συνεχείς δοκιμές καθαρά με βάση την επεξεργαστική ισχύ αλλά παίζει σημαντικό ρόλο κυρίως ο αποθηκευτικός χώρος του σκληρού δίσκου. Το πρώτο στάδιο είναι να αποθηκευτούν κάποια τεράστια αρχεία στον σκληρό δίσκο και αυτή είναι μία αρκετά χρονοβόρα διαδικασία, αφού

αποθηκεύονται συνεχώς πιθανές τιμές μόνο του αριθμού nonce (και δεν γίνονται δοκιμές πάνω σε ολόκληρο το hash) που θα δώσουν την λύση για το επόμενο block (σκεφτείτε ότι πάλι πρέπει να βρεθεί μία τιμή hash 64 ψηφίων δεκαεξαδικής μορφής που αριθμητικά είναι μικρότερη η ίση με ένα δεδομένο hash, το οποίο καθορίζει το νέο block της αλυσίδας σύμφωνα με το block header που αναφέρθηκε στο proof of work. Απλώς, εδώ έχει απομονωθεί η εύρεση του αριθμού nonce στην διαδικασία και το υπόλοιπο μέρος του hash θεωρείται σταθερά, χωρίς να εμπλέκεται στους υπολογισμούς). Όταν ολοκληρωθεί αυτό ακολουθεί η φάση του mining και πρακτικά αυτό που γίνεται είναι συνεχείς υπολογισμοί με βάση τις αποθηκευμένες τιμές στον σκληρό δίσκο. Όσο μεγαλύτερος ο χώρος που διατέθηκε στον σκληρό δίσκο τόσο περισσότερες οι πιθανές τιμές, άρα μεγαλύτερη πιθανότητα να βρεθεί μία λύση του προβλήματος για την προσθήκη του επόμενου κουτιού. Ο κόμβος που θα κάνει επιτυχώς την προσθήκη του κουτιού λαμβάνει την σχετική ανταμοιβή σε κρυπτονομίσματα.

Με απλά λόγια, αυτή η μέθοδος είναι μία παραλλαγή του proof of work στην οποία δεν χρειάζονται πλέον ακραίες τιμές επεξεργαστικής δύναμης αφού οι υπολογισμοί δεν γίνονται σε πραγματικό χρόνο πάνω σε hash values 64 ψηφίων, αλλά στόχος είναι η εύρεση μόνο ενός αριθμού που όταν συνδυαστεί με το hash να δώσει την λύση. Στην τρέχουσα περίοδο θεωρείται αρκετά οικονομικός τρόπος, ωστόσο δεν είναι απίθανο να αυξηθούν σε τρελά επίπεδα οι δυνατότητες των σκληρών δίσκων και να προκύψει ένα αντίστοιχο θέμα του mining με τις εξωφρενικές απαιτήσεις σε κάρτες γραφικών στο proof of work.

- Proof of Burn / PoB: Η ονομασία προέρχεται από το γεγονός ότι οι χρήστες σε ένα blockchain που λειτουργεί με proof of burn, πρέπει να *κάψουν,* ήτοι να σπαταλήσουν κάποια από τα κρυπτονομίσματα τους για να μπορέσουν να γίνουν miners των επόμενων blocks. Αφ΄ενός, η διαδικασία αυτή καθ'αυτή με την οποία επιλέγεται κάποιος miner είναι τυχαία και αξιόπιστη, αλλά αφ΄ετέρου όσο περισσότερα κρυπτονομίσματα ξοδεύουν οι χρήστες τόσο μεγαλύτερη είναι η

συμμετοχή τους μέσα στο σύνολο των χρηστών, από τους οποίους θα επιλεγεί εντέλει κάποιος ως miner. Φανταστείτε, ότι πρόκειται για την αγορά λαχνών με τους οποίους πρόκειται να γίνει κάποια κλήρωση και είναι απολύτως λογικό, πως αν κάποιος έχει αγοράσει πολλούς λαχνούς έχει και μεγαλύτερη πιθανότητα να είναι ο νικητής της κλήρωσης, όμως δεν αποκλείεται και κάποια στιγμή να κερδίσει κάποιος που έχει πάρει για παράδειγμα 1 λαχνό στους 100. Επομένως, πρόκειται για ένα σύστημα που ευνοεί αυτούς οι οποίοι είναι προνομιούχοι και έχουν τη δυνατότητα να ξοδεύουν συνεχώς κρυπτονομίσματα.

- Practical Byzantine Fault Tolerance / pBFT: Για να γίνει κατανοητό πως λειτουργεί ο αλγόριθμος practical byzantine fault tolerance, πρέπει πρώτα να περιγραφεί η έννοια byzantine fault tolerance, δηλαδή η βυζαντινή ανοχή σε σφάλματα όπως είναι η ακριβής μετάφραση. Πρόκειται για κάτι που χαρακτηρίζει ένα σύστημα διασυνδεδεμένων κόμβων, σαν αυτό που λειτουργεί με την τεχνολογία blockchain και αναφέρεται στο γεγονός ότι μπορεί να πετυχαίνεται η συναίνεση μεταξύ των κόμβων, ακόμα και όταν υπάρχουν προβλήματα επικοινωνίας στο δίκτυο. Στη συγκεκριμένη περίπτωση ισχύει πως όλοι οι κόμβοι ψηφίζουν και το αποτέλεσμα προκύπτει από την πλειοψηφία, όμως το πρόβλημα είναι τι γίνεται αν κάποιοι κόμβοι δεν καταγράψουν την ψήφο τους έγκαιρα (υπάρχει χρονικό όριο) ή η υποτιθέμενη ψήφος τους περιέχει άσχετες με το θέμα πληροφορίες πιθανώς ηθελημένα.

  Στον αλγόριθμο practical byzantine fault tolerance οι παραπάνω κόμβοι θεωρούνται ελαττωματικοί και αφαιρούνται από το σύνολο της ψηφοφορίας, ώστε το αποτέλεσμα να βγαίνει μόνο από τις έγκυρες ψήφους. Η γενική παραδοχή είναι ότι για να λειτουργήσει το σύστημα πρέπει πάνω από τα 2/3 των κόμβων να παρουσιάζουν φυσιολογική συμπεριφορά, δηλαδή ακόμα και αν σχεδόν το 1/3 δεν δίνει έγκυρες ψήφους το δίκτυο μπορεί να συνεχίσει να δουλεύει κανονικά. Σε κάθε διαδικασία συναίνεσης που *τρέχει* στο δίκτυο υπάρχει πάντα ένας κύριος κόμβος και οι υπο-κόμβοι, μετά από δοκιμές που γίνονται για να αφαιρεθούν οι ελαττωματικοί κόμβοι. Για παράδειγμα, ο κύριος κόμβος

μεταφέρει το θέμα ψηφοφορίας προς όλους τους υπόλοιπους π.χ. αν μία συναλλαγή είναι έγκυρη για να μπει στο επόμενο κουτί της αλυσίδας και προκύπτει το αποτέλεσμα. Το πρωτόκολλο αυτό αποτελεί κυρίως τον τρόπο που αποφασίζεται πως θα συμπεριφερθεί το δίκτυο για το αν κάποιο transaction είναι έγκυρο όταν υπάρχουν θέματα με προβληματικούς κόμβους, δηλαδή πρόκειται πιο πολύ για πολιτική ασφαλείας παρά για μοντέλο συναίνεσης σχετικά με το πως θα δημιουργηθεί το επόμενο κουτί του blockchain. Για αυτό τον λόγο συναντάται συνδυαστικά με το proof of work ή το proof of stake.

2.3.2.4 Block

Με κάποιον από τους παραπάνω τρόπους συναίνεσης έχει πλέον προστεθεί το νέο κουτί στο blockchain, ωστόσο το εσωτερικό και η δομή του δεν είναι ακριβώς όπως περιγράφηκε σύμφωνα με την Εικόνα 2 καθώς εκεί αναφέρθηκε περιληπτικά τι περιλαμβάνει ένα κουτί για να έχει μία γενική ιδέα ο αναγνώστης. Σε αυτό το σημείο, υπάρχουν αρκετές πληροφορίες και το υπόβαθρο για να κατανοηθεί λεπτομερώς τι γίνεται μέσα σε ένα block. Επομένως, το block χωρίζεται γενικά σε 2 κομμάτια α) στο block header και β) στο block body που περιέχει όλα τα transactions.

** Προσοχή, καθώς αυτή η ανάλυση είναι γενικευμένη και απλώς αποτελεί την βάση για μετάβαση σε διαφοροποιημένα συστήματα blockchain. Πρακτικά, τα κουτιά σε κάθε blockchain είναι αρκετά πιθανό να μην περιέχουν κάποια από τα παρακάτω τμήματα ή να τα έχουν διαφοροποιήσει, προκειμένου να λειτουργούν κάθε φορά με τον επιθυμητό τρόπο. Για παράδειγμα τα nonce και difficulty δεν υπάρχουν σε περιπτώσεις που το blockchain δεν λειτουργεί με κρυπτονομίσματα ή με proof of work, ενώ το merkle root hash πολλές φορές αποθηκεύεται αλλού.**

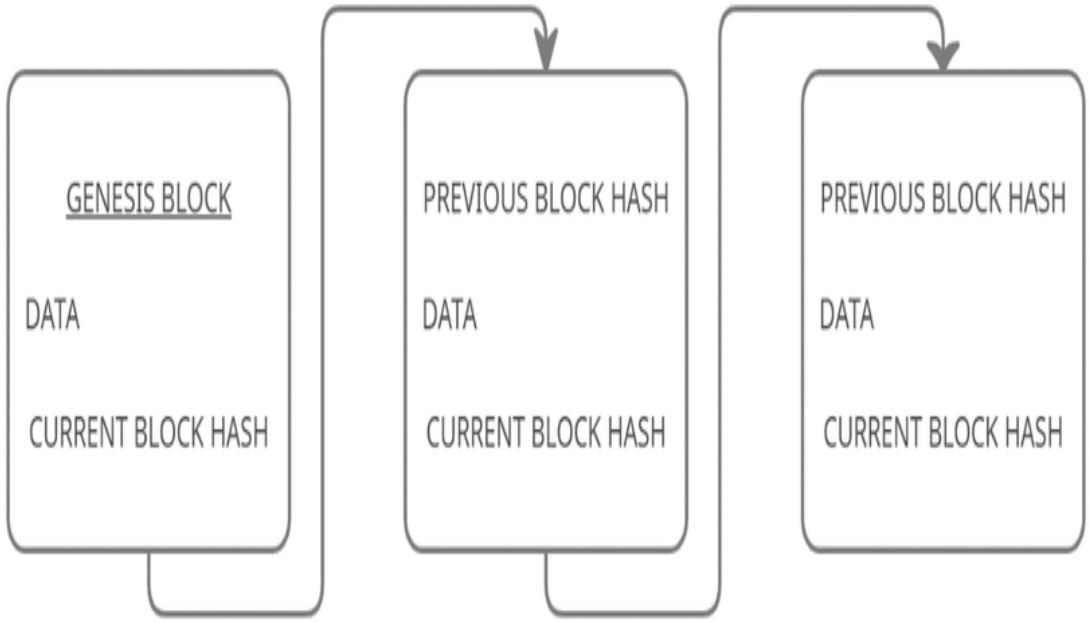
Έχει ήδη αναφερθεί η έννοια block header πολύ συνοπτικά στα μοντέλα συναίνεσης και συγκεκριμένα περιέχει:

- previous block hash: Είναι η κρυπτογραφημένη μορφή των δεδομένων (όλων των συναλλαγών) του προηγούμενου block και συνδέει το τρέχον κουτί με το προηγούμενο.

- current block hash: Είναι η κρυπτογραφημένη μορφή των δεδομένων (όλων των συναλλαγών) του τρέχοντος block.
- merkle root hash: Σύμφωνα με την λογική των Merkle Trees, που περιγράφηκε και στην Εικόνα 1 (σελ. 13) οι συναλλαγές χωρίζονται σε ζευγάρια και κρυπτογραφούνται τα δεδομένα τους με αλγόριθμο hash. Έπειτα, τα hashes χωρίζονται πάλι σε ζευγάρια και κρυπτογραφούνται εκ νέου και αυτό επαναλαμβάνεται μέχρι να προκύψει μόνο ένα hash, το οποίο ονομάζεται merkle root hash.
- block id: Ένας μοναδικός αριθμός που χαρακτηρίζει το συγκεκριμένο block μέσα σε όλη την αλυσίδα.
- block number / version: Συνήθως αναφέρεται συνδυαστικά με το block id, ωστόσο σε πολλές περιπτώσεις είναι ξεχωριστό και προσδιορίζει την έκδοση του blockchain συστήματος κατά την οποία δημιουργήθηκε.
- difficulty: Προσδιορίζει την δυσκολία που θα συναντήσουν οι χρήστες στο mining ενός νέου block και σχετίζεται με το target hash. Το target hash αποτελεί το όριο που δεν πρέπει να ξεπεράσει η αριθμητική αναπαράσταση ενός κατακερματισμένου block header. Ουσιαστικά ο miner πρέπει να βρεί με δοκιμές μία τιμή hash 64 ψηφίων δεκαεξαδικής μορφής που αριθμητικά είναι μικρότερη η ίση με ένα δεδομένο hash (target hash), παίρνοντας κάθε φορά τα δεδομένα του block header, αλλάζοντας μόνο την τιμή του nonce και τρέχοντας τον αλγόριθμο hash. Με απλά λόγια, το difficulty είναι ένας αριθμός που δίνει πόσα ψηφία πρέπει να μεταβληθούν στο hash που τεστάρουν οι χρήστες, ώστε να γίνει το mining και ρυθμίζεται αυτόματα για να είναι σταθερός ο μέσος χρόνος προσθήκης νέων blocks.
- nonce: Όπως αναφέρθηκε και στα μοντέλα συναίνεσης proof of work, proof of capacity είναι ένας αριθμός που αρχικά μεταβάλλεται συνεχώς, ώστε να βρεθεί το σωστό target hash (mining) του κουτιού που θα οδηγήσει τελικά στην προσθήκη του στο blockchain. Εφόσον το κουτί προστεθεί, τότε αυτός ο αριθμός δείχνει ποιά ήταν η μοναδική τιμή του nonce που οδήγησε στο επιτυχές mining.

- timestamp: Η χρονική στιγμή που το κουτί προστέθηκε στην αλυσίδα. Προφανώς κάθε κουτί αντιστοιχεί σε timestamp που είναι αργότερα από το προηγούμενο κουτί. Δηλαδή όλα τα κουτιά στο blockchain είναι ταξινομημένα με βάση το timestamp, ξεκινάμε από το genesis και έπειτα κάθε κουτί αντιστοιχεί σε μεταγενέστερη χρονική στιγμή από το προηγούμενο.

Στην τρέχουσα εργασία δεν χρησιμοποιείται σύστημα blockchain με κρυπτονομίσματα ούτε μοντέλο συναίνεσης proof of work, proof of capacity. Όπως θα αναλυθεί και στο *κεφάλαιο 4* δεν θα περιλαμβάνονται στο κάθε block κανένα από τα nonce, difficulty αλλά ούτε και τα merkle root hash, block number που παράγονται αυτόματα από το δίκτυο BigchainDB. Κάθε block θα περιέχει current block hash, previous block hash και το data, που θα περιέχει διάφορα δεδομένα. Στην υλοποίηση της εφαρμογής μας κάθε block είναι ένα μόνο transaction, άρα οι ιδιότητες block Id, timestamp θα ταυτίζονται με τις αντίστοιχες ιδιότητες της συναλλαγής. Σε γενικές γραμμές, τα blocks θα έχουν προσεγγιστικά την μορφή της παρακάτω εικόνας.



Εικόνα 5 - Τροποποιημένη δομή των blocks για την παρούσα εργασία

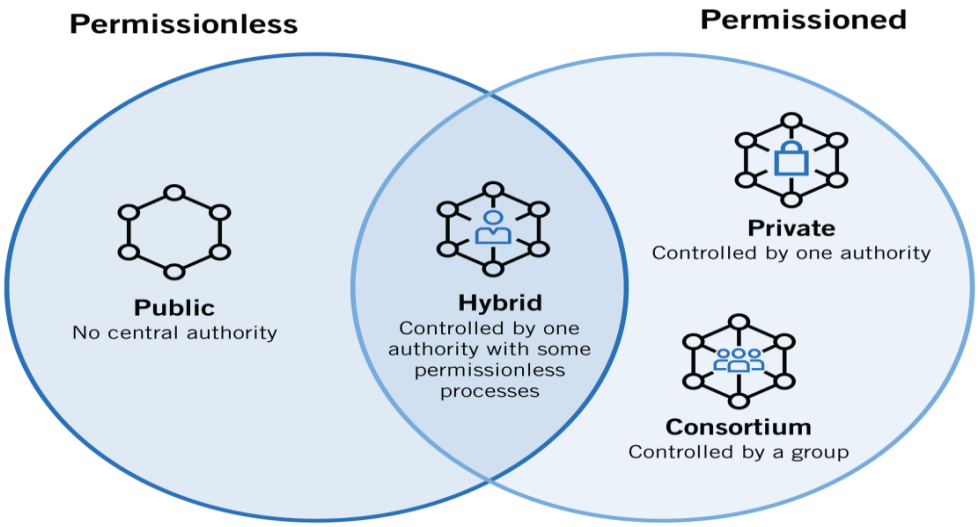
2.3.3 Κατηγορίες Blockchain

Μέχρι στιγμής η τεχνολογία blockchain μπορεί να εφαρμοστεί πάνω σε 4 κύριες κατηγορίες συστημάτων, άρα τα δίκτυα που λειτουργούν με blockchain θα ανήκουν σε κάποιο από τα παρακάτω μοντέλα ή σε συνδυασμούς αυτών:

1. Public Blockchain: Συνήθως χαρακτηρίζονται και από την ονομασία permissionless, καθώς σε αυτές τις αλυσίδες μπορεί οποιοσδήποτε χρήστης να εισέλθει και να γίνει κόμβος του δικτύου, χωρίς να απαιτείται κάποιου είδους άδεια ή έγκριση. Το δημόσιο blockchain χρησιμοποιείται κατά κύριο λόγο σε συστήματα με κρυπτονομίσματα και η διαχείριση του γίνεται συγκεντρωτικά από όλους τους κόμβους, σύμφωνα με τις διαδικασίες συναίνεσης που περιγράφηκαν.
2. Private Blockchain: Στο ιδιωτικό ή αλλιώς permissioned blockchain, χρειάζεται έγκριση για να μπορέσει κάποιος να εγγραφεί στο δίκτυο και να πραγματοποιήσει διάφορες ενέργειες. Η απόφαση για το αν θα εισέλθει στο σύστημα ένας χρήστης που κάνει αίτηση, λαμβάνεται από την εκάστοτε εταιρεία που δημιούργησε το blockchain και έχει το ρόλο του διαχειριστή. Ένας χρήστης που θα καταφέρει να μπει στο ιδιωτικό blockchain δεν αποτελεί αυτόματα ισότιμο κόμβο με τους υπόλοιπους του δικτύου και οι επιτρεπόμενες ενέργειες του καθορίζονται από τον διαχειριστή. Επομένως, σε αυτή την κατηγορία blockchain υπάρχει μία κεντρική οντότητα που θα πρέπει να δώσει την άδεια-permission για το αν και πότε θα γίνει κάποιος μέρος του δικτύου και ελέγχει ανά πάσα στιγμή ποιά θα είναι η συμπεριφορά του κάθε κόμβου σε αυτό το δίκτυο, για παράδειγμα αν θα επικυρώνει συναλλαγές (δικαιώματα κόμβου).
3. Hybrid Blockchain: Σε αυτή την υβριδική μορφή χρησιμοποιούνται χαρακτηριστικά από αμφότερα τα παραπάνω μοντέλα και υπάρχουν αρκετοί διαφορετικοί συνδυασμοί. Αναφορικά, 2 από τις πολλές εφαρμογές του υβριδικού μοντέλου είναι α) μία εταιρεία υλοποιεί ένα τέτοιο blockchain στο οποίο μπορεί να εισέλθει οποιοσδήποτε ελεύθερα και να πραγματοποιήσει συναλλαγές, αλλά δεν μπορεί από μόνος του να αποφασίζει για το αν κάποιο block θα προστεθεί στην

αλυσίδα (αυτό ελέγχεται από την εταιρεία) ή β) η εταιρεία πρέπει να δώσει έγκριση για να μπει κάποιος σαν κόμβος στο blockchain, αλλά εφόσον μπει έχει τα ίδια δικαιώματα με όλους τους υπόλοιπους χρήστες για επικύρωση συναλλαγών και συμμετοχή στη συναίνεση π.χ. mining (αν ακολουθείται το proof of work). Εν κατακλείδει, κάποιος που αναπτύσσει μία εφαρμογή με blockchain έχει την επιλογή να τροποποιήσει την λειτουργία του παίρνοντας στοιχεία από διάφορα μοντέλα και μία τέτοια προσέγγιση υλοποιείται στην παρούσα μελέτη.

4. Consortium Blockchain: Η συγκεκριμένη κατηγορία αποτελεί μία μικρή παραλλαγή του υβριδικού blockchain. Η μοναδική διαφορά είναι πως δεν υπάρχει μόνο μία κεντρική οντότητα που ελέγχει την λειτουργία της αλυσίδας, αλλά υπάρχουν πολλαπλές εταιρείες που έχουν το ρόλο του διαχειριστή και συγκεντρωτικά αποφασίζουν για διάφορα θέματα, όπως ποιοί κόμβοι θα εισέλθουν στο δίκτυο, ποιοί κόμβοι θα έχουν παραπάνω δικαιώματα και ποιοί λιγότερα, πως θα ελέγχονται οι συναλλαγές κλπ. Ουσιαστικά, είναι μία προσπάθεια να μετατραπεί το υβριδικό μοντέλο σε κάτι πιο αποκεντρωμένο (δεν ελέγχονται όλοι οι κόμβοι από μία μόνο εταιρεία, αλλά κάθε εταιρεία ελέγχει ένα ποσοστό τους), όμως ταυτόχρονα τείνει προς το permissioned blockchain επειδή ανάμεσα στο πλήθος των διαχειριστών πιθανώς να μην υπάρχει τόσο μεγάλη εμπιστοσύνη και αυτό οδηγεί σε αυξημένους ελέγχους.



Εικόνα 6 - Κατηγορίες Blockchain

2.4 Εφαρμογές

Μολονότι δεν έχει περάσει μεγάλο χρονικό διάστημα από την πρώτη ανάπτυξη ολοκληρωμένου συστήματος με blockchain (2009 bitcoin) και κυρίως από την συνειδητοποίηση πως οι δυνατότητες ξεπερνούν σημαντικά τα κρυπτονομίσματα και την έναρξη ενσωμάτωσης προς άλλες κατευθύνσεις (Ethereum 2015), ήδη οι χρήσεις της τεχνολογίας έχουν επεκταθεί σε αρκετούς τομείς, πλην των κρυπτονομισμάτων, με μεγάλη επιτυχία. Ακολουθεί μία λίστα με μερικές από τις υπάρχουσες υλοποιήσεις της καινοτομίας blockchain, γεγονός που αποδεικνύει τις υψηλότατες προοπτικές που την χαρακτηρίζουν και δικαίως αποτελεί σημαντικό πεδίο έρευνας με προυπολογισμούς που θα ξεπερνούν παγκοσμίως σχεδόν τα 12.5 δισεκατομμύρια δολάρια μέχρι το τέλος του 2022.

- ✓ Υπάρχει πληθώρα διαδικτυακών εφαρμογών και ιστοτόπων που ασχολούνται με συναλλαγές κρυπτονομισμάτων π.χ. bitcoin, ethereum, dogecoin, litecoin κλπ με τα ομώνυμα applications/digital wallets.
- ✓ Πολλές εφαρμογές δεν έχουν δικά τους κρυπτονομίσματα και λειτουργούν ως ενδιάμεση πλατφόρμα αγοραπωλησίας υπαρχόντων κρυπτονομισμάτων, δίνοντας την δυνατότητα να γίνονται συναλλαγές μεταξύ διαφορετικών ειδών με αυτόματη αντιστοίχηση της εικονικής αξίας π.χ. από bitcoin σε ethereum. Τέτοιες εταιρείες είναι το etoro και το coinbase.
- ✓ Συστήματα ηλεκτρονικής ψηφοφορίας με χρήση blockchain για ασφαλή και αξιόπιστη καταχώρηση των ψήφων, επομένως με εγγύηση ότι δεν θα υπάρχει αλλοίωση του αποτελέσματος. Μία εταιρεία που πραγματεύεται αυτή τη διαδικασία είναι η FollowMyVote.
- ✓ Η ίδια λογική της ηλεκτρονικής ψηφοφορίας έχει εφαρμοστεί από οργανισμούς σαν τον Patently Walmart στην αποθήκευση μετρήσεων και ρυθμίσεων από αισθητήρες IoT σε δίκτυο blockchain. Οι χρήστες δημιουργούν λογαριασμό και αποστέλουν τα δεδομένα τους στο blockchain, γνωρίζοντας ότι μόνο αυτοί μπορούν να τα δουν, να τα ελέγξουν ανά πάσα στιγμή και δεν ανησυχούν για πιθανές κακόβουλες τροποποιήσεις.

- ✓ Η βάση δεδομένων blockchain ενσωματώνεται με μεγάλη επιτυχία σε εφαρμογές μηνυμάτων σαν το Secretum (που υποστηρίζει και μεταφορές κρυπτονομισμάτων). Μία τέτοια εφαρμογή εφησυχάζει τους χρήστες πως τα μηνύματα τους προορίζονται μόνο για τον παραλήπτη που επιλέγουν, ότι δεν θα αλλοιωθούν και δεν πρόκειται να διαγραφούν.
- ✓ Οργανισμοί υγείας όπως ο MedicalChain συνεργάζονται με δημόσια νοσοκομεία ή ιδιωτικές κλινικές και παρέχουν τα ευαίσθητα προσωπικά ιατρικά δεδομένα των ασθενών, που βρίσκονται αποθηκευμένα στο blockchain, μόνο με την δική τους συγκατάθεση. Προφανώς, αυτή η διαδικασία αφ'ενός προστατεύει τον ασθενή, τον διαβεβαιώνει ότι οι πληροφορίες του είναι ασφαλείς, συνεχώς διαθέσιμες και ορατές μόνο όταν ο ίδιος το εγκρίνει, αλλά εφ'ετέρου υπάρχει εύκολη και άμεση πρόσβαση σε αυτές.
- ✓ Κυβερνήσεις έχουν υλοποιήσει διάφορες διαδικασίες με αποκλειστική χρήση blockchain ή σκοπεύουν να το κάνουν στο άμεσο μέλλον. Για παράδειγμα η Εσθονία έχει δημιουργήσει ένα ηλεκτρονικό σύστημα ψηφοφορίας με blockchain και το ίδιο ετοιμάζεται να πραγματοποιήσει και η Ουκρανία. Επίσης, η Αυστραλία έχει προνοήσει ώστε οι τουρίστες που επισκέπτονται την χώρα να μπορούν να κάνουν πληρωμές αποκλειστικά με κρυπτονομίσματα.
- ✓ Τραπεζικές επιχειρήσεις έχουν δημιουργήσει το δικό τους ιδιωτικό blockchain για ασφαλή καταχώρηση των πελατειακών δεδομένων ή/και τα δικά τους κρυπτονομίσματα, που είναι πλήρως αποδεκτά για συναλλαγές μεταξύ πελατών της εταιρείας. Χαρακτηριστικά παραδείγματα τέτοιων οργανισμών με παγκόσμια φήμη, αποτελούν οι JP Morgan και Goldman Sachs (με το κρυπτονόμισμα stablecoin USDC).

3. Τεχνολογίες και εργαλεία

3.1 Περιβάλλον ανάπτυξης

Για την ανάπτυξη της παρούσας εφαρμογής χρησιμοποιείται το περιβάλλον Visual Studio Code ή όπως είναι ευρέως γνωστό στο προγραμματιστικό κοινό VS Code. Ανήκει στην κατηγορία προγραμμάτων IDE-Integrated Development Environment τα οποία υποστηρίζουν την ανάπτυξη εφαρμογών σε πάρα πολλές γλώσσες προγραμματισμού, διαθέτουν επεξεργαστή κώδικα, λειτουργίες αποσφαλμάτωσης κώδικα σε πραγματικό χρόνο και ορίζουν αυτόματα τις απαραίτητες ρυθμίσεις για την εκάστοτε υλοποίηση. Το συγκεκριμένο πρόγραμμα επιλέχθηκε για την τρέχουσα εργασία διότι αποτελεί εύχρηστο περιβάλλον για την ανάπτυξη web applications, είναι ελαφρύ (δεν καταναλώνει μεγάλο ποσοστό επεξεργαστικής ισχύος και μνήμης RAM) και ενσωματώνει πληθώρα βιβλιοθηκών και πακέτων που είναι απολύτως απαραίτητα σε τέτοιου είδους προγραμματιστικές διαδικασίες.

3.2 Γλώσσες προγραμματισμού

- Μία αρκετά δημοφιλής γλώσσα προγραμματισμού για την δημιουργία εφαρμογών ιστοτόπου είναι η γλώσσα Javascript/JS που αποτελεί γενικά την επικρατέστερη επιλογή για την ανάπτυξη του front-end ή του GUI-Graphical User Interface, δηλαδή το οπτικό/γραφικό κομμάτι της εφαρμογής με το οποίο αλληλεπιδρά ο χρήστης. Ωστόσο, στην περίπτωση μας η γλώσσα Javascript χρησιμοποιείται επίσης για την ανάπτυξη του back-end (server-side) και για να γίνει η απαραίτητη σύνδεση με τη βάση δεδομένων, καθώς όπως θα αναλυθεί στη συνέχεια η εφαρμογή μας υλοποιείται μέσω του MERN stack.
- Επιπλέον, χρησιμοποιούνται οι γλώσσες HTML και CSS καθώς πρόκειται για εφαρμογή ιστοτόπου και θα γίνει χρήση τους για την εμφάνιση των ιστοσελίδων της εφαρμογής σε έναν φυλλομετρητή. Η γλώσσα HTML-HyperText Markup Language καθορίζει πως θα είναι δομημένη και τι θα περιέχει μία ιστοσελίδα μέσω των html elements (paragraph, heading, input, button, image κλπ), ενώ η CSS-Cascading

Style Sheets προσδιορίζει την εμφάνιση των παραπάνω στοιχείων και κατ'επέκταση ολόκληρης της ιστοσελίδας.

3.3 Εργαλεία ανάπτυξης

Με τον όρο εργαλεία αναφερόμαστε στο σύνολο των βιβλιοθηκών-libraries ή modules, πακέτων-packages, frameworks και περιβαλλόντων που χρησιμοποιούνται στον προγραμματισμό της εφαρμογής.

- βιβλιοθήκη-library: Είναι ένα αρχείο κώδικα με έτοιμες, υλοποιημένες βασικές λειτουργίες.
- πακέτο-package: Είναι ένα σύνολο βιβλιοθηκών, το οποίο περιέχει τουλάχιστον ένα αρχείο.
- framework: Είναι μία έννοια που σχετίζεται άμεσα με τον προγραμματισμό εφαρμογών και αποτελεί τη βάση για να ξεκινήσεις την ανάπτυξη του κώδικα. Ουσιαστικά, ένα framework είναι ένα προσχέδιο/template που περιλαμβάνει ομάδες πακέτων και βιβλιοθηκών και πάνω σε αυτές γίνονται προσθήκες ή τροποποιήσεις για να δημιουργήσει ο προγραμματιστής την δική του εφαρμογή.

Αρχικά, θα χρησιμοποιηθούν 4 εργαλεία σύμφωνα με το MERN stack (stack είναι ένα σύνολο εργαλείων) και αναφορικά με αυτό, πρόκειται για μία προσέγγιση full-stack (αναπτύσσεται το front-end, το back-end μίας εφαρμογής και γίνεται η σύνδεση με τη βάση δεδομένων) που ενσωματώνει τα MongoDB Express React Node.

1. MongoDB-mongoose: Το MongoDB αποτελεί μία δημοφιλή βάση δεδομένων με την τεχνολογία NoSql, δηλαδή δεν χρησιμοποιούνται καθόλου Sql εντολές για να γίνουν οι διάφορες ενέργειες (διάβασμα, επεξεργασία, αποθήκευση, διαγραφή) στις εγγραφές της βάσης. Επιπλέον, δεν υπάρχουν πίνακες, στήλες, σειρές/εγγραφές (όπως στις Sql βάσεις) αλλά συλλογές/collections οι οποίες θα περιέχουν τα έγγραφα/documents, τα οποία με τη σειρά τους θα περιέχουν τα πεδία/fields με τη μορφή ζεύγος key-value π.χ. id: 1. Συμπερασματικά,

το κάθε έγγραφο περιέχει ένα JSON αντικείμενο, που αποτελείται από διάφορα πεδία και κάθε συλλογή περιέχει πολλά έγγραφα. Η βάση δεδομένων MongoDB ανοίγει με το πρόγραμμα MongoDB Compass (όπου μπορούν να γίνουν χειροκίνητα οι ενέργειες) και η υλοποίηση μέσα στον κώδικα μας γίνεται μέσω της βιβλιοθήκης mongoose.js που ενσωματώνεται στην εφαρμογή γράφοντας την εντολή *npm install mongoose* στο terminal του VS Code, άρα οι εντολές με τις οποίες διαχειριζόμαστε τις εγγραφές στη βάση είναι σε γλώσσα Javascript και όχι Sql. Συχνά το MongoDB αναφέρεται ως framework με τη σημασία του συνόλου MongoDB, MongoDB Compass και mongoose.js.

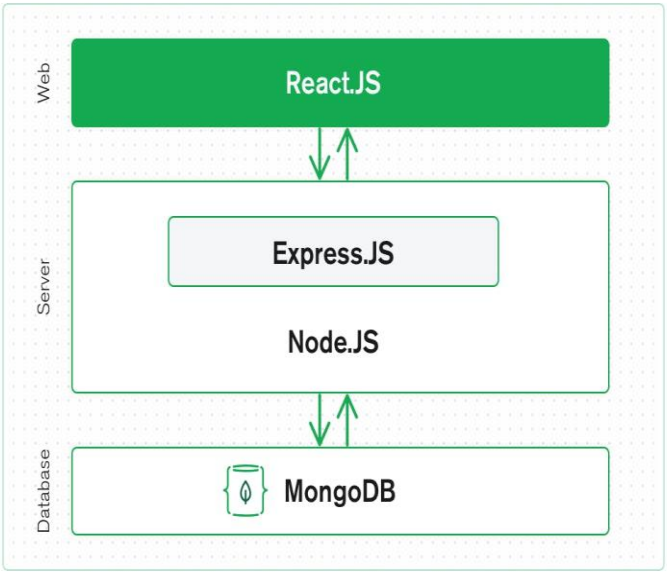
2. Express: Είναι ένα framework που υλοποιεί τον server της εφαρμογής, δηλαδή το back-end και λειτουργεί ως ο ενδιάμεσος για να συνδεθεί το front-end με τη βάση δεδομένων. Πρακτικά, όλα τα αιτήματα-requests που κάνει ο χρήστης στο front-end της εφαρμογής στέλνονται στο back-end, εκεί επεξεργάζονται με τις εντολές javascript που περιλαμβάνει το express.js και στέλνονται στην αντίστοιχη διαδρομή, ανάλογα γίνονται οι ενέργειες στη βάση δεδομένων με το mongoose.js και τέλος επιστρέφονται (πάλι με το express.js) οι απαντήσεις στον χρήστη, ο οποίος ενημερώνεται στο front-end. Το express.js προστίθεται σε μία εφαρμογή μέσω της εντολής *npm install express* στο terminal του VS Code.
3. React: Πρόκειται για ένα front-end javascript library που με το σύνολο λειτουργιών του προσδιορίζει πως θα είναι δομημένο το κομμάτι της εφαρμογής με το οποίο αλληλεπιδρά άμεσα ο χρήστης, το User Interface. Περιλαμβάνει τα components που είναι το ανώτατο επίπεδο δομής σε κάθε ιστοσελίδα της εφαρμογής π.χ. Login Component και αυτά αποτελούνται αντίστοιχα από μικρότερες δομές, ήτοι άλλα components και html elements που μπορεί να είναι buttons, forms, inputs κοκ. Τα components δημιουργούνται με διάφορους παρόμοιους τρόπους (class,function) ανάλογα με την επιθυμητή προγραμματιστική προσέγγιση, για παράδειγμα αν θέλουμε να περάσουμε κάποια παράμετρο-props στο component συνήθως θα το ορίσουμε ως class

ενώ αν θέλουμε απλά να εμφανίζει κάποια στοιχεία θα το ορίσουμε ως function (πλέον ο προγραμματιστής αποφασίζει ποιά μέθοδο θα ακολουθήσει ανάλογα με το στυλ του). Για να προστεθούν τα html elements του κάθε component χρησιμοποιείται το JSX, μία παραλλαγή της HTML που συνδυάζει και την γλώσσα javascript π.χ. για να εμφανιστούν html elements μέσα σε for loop.

Η βιβλιοθήκη εγκαθίσταται αυτόματα όταν, όπως θα δούμε παρακάτω, δημιουργείται το front-end κομμάτι της εφαρμογής μέσω του create-react-app και προστίθεται στον κώδικα με την εντολή *import React from 'react'* και συνοδεύεται από το πακέτο react-dom, το οποίο επίσης ενσωματώνεται αυτόματα και προστίθεται με *import ReactDOM from 'react-dom'*. Το ReactDOM εμφανίζει τα views που δημιουργούνται με το react και βασίζεται στο Document Object Model. Υπάρχει ένα γενικό html document συνήθως *index.html* στον φάκελο *public* το οποίο μέσα στο *body* περιέχει το * * και εκεί περιέχεται το υψηλότερο επίπεδο component της εφαρμογής μας συνήθως το *App component*. Το *App* εμφανίζεται μέσα στο *root* με την εντολή *ReactDOM.render(<App />, document.getElementById('root'))* και έπειτα διαιρείται σε δενδροειδή μορφή προς τα κάτω σε όλα τα υπόλοιπα components/views και τα επιμέρους html elements, τα οποία μπορεί να χωρίζονται σε επιμέρους components ή elements κοκ.

4. Node: Αποτελεί ένα javascript runtime, ένα περιβάλλον στο οποίο τρέχει/εκτελείται ο κώδικας javascript χωρίς να χρειάζεται η εκτέλεση του σε κάποιο φυλλομετρητή. Η δυνατότητα αυτή υπάρχει, γιατί το node.js προσομοιώνει με ένα runtime engine (V8 engine, αντιστοιχίζει τον κώδικα σε γλώσσα κατανοητή από τον υπολογιστή) τον τρόπο που θα έτρεχε ο κώδικας π.χ. στο Google Chrome με αποτέλεσμα να μπορεί ο προγραμματιστής να δημιουργήσει την εφαρμογή του και να την τρέξει από την γραμμή εντολών πληκτρολογώντας π.χ. *node myapplication.js*. Σημαντικό κομμάτι της τεχνολογίας node.js αποτελεί το npm-node package manager με το οποίο γίνεται εγκατάσταση διάφορων modules ή πακέτων στην εφαρμογή μέσω της εντολής *npm install ...* ή τρέχει ένα

συγκεκριμένο αρχείο που έχει οριστεί ως αρχή/start για να ξεκινήσει η εφαρμογή *npm start*. Το node.js και το npm εγκαθίστανται στον υπολογιστή, κατεβάζοντας και τρέχοντας το σχετικό αρχείο από *https://nodejs.org/en/download/*.



Εικόνα 7 - MERN stack

Ακολουθεί μία λίστα με όλα τα modules/libraries και τα πακέτα που ενσωματώθηκαν στην εφαρμογή.

- bigchaindb-driver: Πρόκειται για το πακέτο, που εγκαθίσταται στην εφαρμογή πληκτρολογώντας στο terminal του VS Code την εντολή *npm install bigchaindb-driver* και περιέχει όλους τους απαραίτητους οδηγούς για να γίνει η διασύνδεση με το δίκτυο blockchain BigchainDB και να λειτουργήσει ο τοπικός μας υπολογιστής ως κόμβος σε αυτό το δίκτυο.

  Το BigchainDB είναι ο συνδυασμός μίας MongoDB βάσης δεδομένων και ενός blockchain μηχανισμού που ονομάζεται Tendermint. Ουσιαστικά, το λογισμικό Tendermint είναι αυτό που 'κλωνοποιεί' και κατανέμει την εκάστοτε εφαρμογή blockchain (στέλνει τα ίδια αντίγραφα της εφαρμογής) σε διάφορους κόμβους μέσα στο παγκόσμιο δίκτυο του και επιπλέον διαχειρίζεται τη διαδικασία συναίνεσης με proof of stake. Με αυτό τον τρόπο το BigchainDB έχει πάρει μία NoSql

βάση δεδομένων MongoDB μαζί με κάποιον κώδικα και την έχει μετατρέψει σε blockchain μέσω της τεχνολογίας Tendermint, με αποτέλεσμα να προκύψει μία NoSql βάση δεδομένων με λειτουργικότητα blockchain. Εμείς δεν αλληλεπιδρούμε άμεσα με τον αλγόριθμο Tendermint καθώς είναι ενσωματωμένο μέσα στο BigchainDB, ωστόσο αυτό που πρέπει να γνωρίζουμε είναι ότι οποιοσδήποτε προγραμματιστής μπορεί να αναπτύξει την εφαρμογή του και εισάγοντας την στο Tendermint υλοποιείται αυτόματα το δικτυακό κομμάτι και η συναίνεση, πρακτικά μεταμορφώνοντας την σε blockchain application.

Η προσέγγιση που ακολουθείται στην παρούσα εργασία είναι η εξής:

- Αρχικά, γίνεται η εγκατάσταση του Docker* στον υπολογιστή μας και μέσω αυτού αποθηκεύουμε όλα τα αρχεία που χρειάζονται για να τρέξει ο υπολογιστής μας το BigchainDB instance (με τις απαραίτητες ρυθμίσεις) και να γίνει κόμβος του δικτύου του. Μέσα από το πρόγραμμα Docker Desktop ξεκινάμε το container που δημιουργήθηκε.
- Έπειτα, έχουμε την εφαρμογή μας που υλοποιεί blockchain λειτουργικότητα, ενώ αποθηκεύει τα δεδομένα της σε μία τοπική βάση MongoDB. Μέσα στον κώδικα μας γίνεται η σύνδεση με το BigchainDB στο οποίο ανήκουμε ως κόμβος και επικοινωνούμε με αυτό.
- Τα μηνύματα που ανταλλάσονται μεταξύ των χρηστών της εφαρμογής μας, στέλνονται ως συναλλαγές κρυπτογραφημένα (1 μήνυμα = 1 συναλλαγή) στο BigchainDB και αποθηκεύονται μόνιμα στο blockchain του με όλες τις σχετικές πληροφορίες. Τα ίδια μηνύματα αποθηκεύονται ως κουτιά (1 μήνυμα = 1 συναλλαγή = 1 block) με τις σχετικές πληροφορίες στο τοπικό μας blockchain που υπάρχει στην τοπική μας βάση MongoDB.
- Κάθε φορά που τρέχει η εφαρμογή γίνονται διάφοροι έλεγχοι για την ακεραιότητα του τοπικού μας blockchain και αν διαπιστωθεί το οποιοδήποτε πρόβλημα, τότε γίνεται ανακατασκευή του

επαναφέροντας τις σωστές τιμές (διαγράφουμε το τοπικό blockchain, τραβάμε τα δεδομένα από το BigchainDB και ξαναδημιουργούμε το δικό μας blockchain).

- Με αυτό τον τρόπο έχουμε πετύχει να δημιουργήσουμε το δικό μας blockchain στην εφαρμογή μηνυμάτων με όλα τα σωστά χαρακτηριστικά. Όλα τα κουτιά στο blockchain μας υπάρχουν σαν έγγραφα στη δική μας βάση MongoDB αλλά ταυτόχρονα υπάρχουν ασφαλή σαν συναλλαγές και στο blockchain του BigchainDB. Επομένως, η εφαρμογή μας και το blockchain της έχει αποκεντρωμένο (τα κουτιά του blockchain μας δεν διαχειρίζονται από μία οντότητα αλλά από όλους τους κόμβους του BigchainDB. Εισέρχονται πρώτα σαν συναλλαγές στο BigchainDB, εκεί γίνεται η διαδικασία συναίνεσης, στην οποία συμμετέχουμε ως κόμβος, με βάση το Tendermint για να προστεθεί το κουτί που την περιέχει και αφού προστεθεί, τότε και μόνο τότε προστίθεται στο blockchain μας), κατανεμημένο (το blockchain μας, δηλαδή το σύνολο των κουτιών υπάρχουν σαν συναλλαγές σε όλους τους κόμβους του δικτύου BigchainDB) και αμετάβλητο χαρακτήρα (όπως αναφέραμε, αν γίνει αλλοίωση του blockchain μας γίνεται ανακατασκευή του με τις σωστές τιμές και άρα οποιαδήποτε τροποποίηση αναιρείται, ενώ το ίδιο πράγμα συμβαίνει και με τα κουτιά στην αλυσίδα BigchainDB).
- Έχουμε δημιουργήσει μία υβριδική περίπτωση αλυσίδας η οποία αποκτά τα κύρια χαρακτηριστικά που την προσδιορίζουν ως blockchain και μέσω της διασύνδεσης με το BigchainDB, αλλά και από τον δικό μας κώδικα. Συνοπτικά λοιπόν, πρόκειται για μία περίπτωση στην οποία όλα τα κουτιά ενός blockchain περιέχονται ως συναλλαγές σε ένα άλλο blockchain.

* Σε αυτό το σημείο πρέπει να εξηγηθούν κάποιες βασικές έννοιες γύρω από την τεχνολογία Docker, καθώς χρησιμοποιείται ως αφετηρία για την χρήση του BigchainDB.

- Container: Αποτελεί ένα εκτελέσιμο πακέτο λογισμικού, στο οποίο υπάρχουν ενσωματωμένα ο κώδικας της εφαρμογής μαζί με όλες τις αναφορές, τις εξαρτήσεις, τα εργαλεία συστήματος και τις απαραίτητες βιβλιοθήκες, που απαιτούνται για να τρέξει σε οποιοδήποτε υπολογιστικό περιβάλλον.
- Η κύρια ιδέα γύρω από το Docker, είναι να τοποθετηθούν οι διάφορες ανεξάρτητες διαδικασίες μίας εφαρμογής στο εσωτερικό εκτελέσιμων πακέτων λογισμικού (containers), με αποτέλεσμα να υπάρχει ανεξαρτησία από τις εκάστοτε υποδομές και να μπορεί να πραγματοποιηθεί η εκτέλεση τους σε οποιοδήποτε λειτουργικό σύστημα. Τα docker images είναι αρχεία που τα εκτελούμε για να δημιουργηθούν τα containers και με την εντολή *docker pull bigchaindb/bigchaindb:all-in-one*, κατεβάζουμε από το online αποθετήριο εικόνων Docker Hub την εικόνα bigchaindb:all-in-one και την τρέχουμε στο terminal με την εντολή *docker run -d --name bigchaindb -p 9984:9984 -p 9985:9985 -p 26017:27017 -p 26657:26657 bigchaindb/bigchaindb:all-in-one,* με αποτέλεσμα να δημιουργηθεί και να τρέξει στο background το container (-d) με όνομα bigchaindb (--name bigchaindb) και οι θύρες του υπολογιστή μας 9984, 9985, 26017, 26657 αντιστοιχίζονται στις θύρες του container 9984, 9985, 27017, 26657 με την εντολή -p (publish).

  Η αντιστοίχηση των θυρών δημιουργεί τη σύνδεση του τοπικού μας υπολογιστή με το BigchainDB και συγκεκριμένα οι θύρες 9984, 9985 θα συνδέονται με τους 2 BigchainDB σέρβερς που διαχειρίζονται τα αιτήματα στο API (9984) *http://localhost:9984/api/v1/* και την αποστολή ενημερωτικών απαντήσεων από το BigchainDB όταν επικυρώθηκε μία συναλλαγή (9985), η 26017 συνδέεται με τη βάση MongoDB του BigchainDB δικτύου (27017) και η 26657 με τον σέρβερ του Tendermint ώστε να γίνεται η επικοινωνία για να προκύπτει το

αποτέλεσμα από τη διαδικασία συναίνεσης, να στέλνονται οι συναλλαγές στους υπόλοιπους κόμβους του blockchain και να επιβεβαιώνεται γενικά η ύπαρξη του κόμβου αυτού στο δίκτυο (26657).

Συνεχίζουμε με τα πακέτα και τις βιβλιοθήκες, που εγκαθίστανται στην εφαρμογή μέσω των εντολών που γράφουμε στο terminal του VS Code, ενώ βρισκόμαστε στο αρχικό directory της εφαρμογής μας και προστίθενται στον κώδικα του front-end με την εντολή import ('...') ή του back-end με την εντολή require('...'):

- create-react-app: Πρόκειται για ένα πακέτο που γίνεται εγκατάσταση με την εντολή *npm install create-react-app*. Πρώτα πληκτρολογούμε *npm install npx* και έπειτα μέσω της εντολής *npx create-react-app Frontendui* δημιουργείται το template της εφαρμογής με όνομα Frontendui, που θα χρησιμοποιεί react για το UI. Επομένως, πλέον θα υπάρχει ένας φάκελος με το όνομα Frontendui στον οποίο περιέχονται βασικά αρχεία, υπάρχει προεγκατεστημένο το react library και μπορούμε να ξεκινήσουμε να προγραμματίζουμε το front-end της εφαρμογής μας πάνω στο react. Ενημερωτικά, το npx (node package execute) είναι ένας τρόπος να εκτελεστούν πακέτα που υπάρχουν έτοιμα στο μητρώο του npm.
- react-router-dom: Είναι ένα πακέτο, το οποίο προστίθεται στον φάκελο της εφαρμογής μας με την εντολή *npm install react-router-dom* και περιέχει όλα τα αρχεία για να μπορεί ο χρήστης να χρησιμοποιήσει στον κώδικα του την βιβλιοθήκη React Router. Αυτή η βιβλιοθήκη επιτρέπει την προσθήκη components π.χ. Switch, Route, Link με τα οποία γίνονται μεταβάσεις από μία ιστοσελίδα της εφαρμογής σε μία άλλη (από το ένα view σε κάποιο άλλο) και προσφέρει διάφορες δυνατότητες, όπως για παράδειγμα να καθοριστεί το ακριβές url του κάθε view, τι παραμέτρους θα δέχεται το κάθε view ή που θα μεταφέρεται ο χρήστης όταν πατάει κάποιο κουμπί.

- bootstrap: Το bootstrap είναι ένα δημοφιλές framework για το user interface που χρησιμοποιεί έτοιμες κλάσεις σε html elements, οι οποίες αντιστοιχούν σε προ-υλοποιημένα css στυλ εμφάνισης. Για παράδειγμα αν κάποιος κάνει ως γνωστόν εγκατάσταση το bootstrap με την εντολή *npm install* bootstrap, μετά έχει την δυνατότητα να επιλέξει ανάμεσα σε πολλές βιβλιοθήκες και να προσθέσει κάποια από αυτές σε ένα αρχείο js. Μία από τις πιο γνωστές βιβλιοθήκες είναι το μικρό σε μέγεθος (minified) module *bootstrap.min.css,* που περιέχει τέτοια έτοιμα στυλ εμφάνισης css και αυτό χρησιμοποιείται στην παρούσα εργασία. Η ενσωμάτωσή του στον κώδικα react.js (όπως και όλα τα πακέτα ή libraries στο front-end) γίνεται με την εντολή *import '../node_modules/bootstrap/dist/css/bootstrap.min.css'*, όπου μέσα στα εισαγωγικά περιέχεται το σχετικό path του module που αντιστοιχεί στη θέση του μέσα στον φάκελο Frontendui του react project που δημιουργήσαμε αρχικά. Η χρήση του γίνεται λοιπόν μέσα στον JSX κώδικα του αρχείου με έτοιμες κλάσεις όπως π.χ. *<ul className="navbar-nav">...<ul>*, που αντιστοιχεί στη λίστα αντικειμένων που θα υπάρχουν σε μία μπάρα περιήγησης στο τρέχον view. Η μπάρα θα έχει οριστεί πιο πάνω πάλι με κλάση boostrap και θα περιέχει στο εσωτερικό της το ul, ωστόσο περισσότερες λεπτομέρειες θα φανούν στην παράθεση κώδικα του κεφαλαίου 4.
- sweetalert2: Η εγκατάσταση του πακέτου γίνεται πληκτρολογώντας *npm install sweetalert2* και αποτελεί μία αντίστοιχη περίπτωση του Toast πακέτου που πιθανώς να γνωρίζετε από την γλώσσα java. Χρησιμοποιείται για την δημιουργία pop-up παραθύρων που εμφανίζονται στην οθόνη για ένα χρονικό διάστημα και έχουν στόχο να ενημερώνουν το χρήστη για το αποτέλεσμα κάποιας ενέργειας που έκανε μέσα στην εφαρμογή, για παράδειγμα να πληροφορήσει ότι έγινε κάποιο redirect *showMessage('info', 'You were redirected here because you pressed the button Back')* ή ότι έγινε με επιτυχία το login *showMessage('success', 'Your login was successful.')* ή πως ο κωδικός

πρόσβασης ήταν λάθος *showMessage('error', 'The password you typed is incorrect.')*.

- jwt-then: Η λειτουργία αυτού του πακέτου περιλαμβάνει την υπογραφή/δημιουργία ενός JSON Web Token που είναι τελικά ένα string σε κρυπτογραφημένη μορφή, το οποίο έχει προκύψει από το μοναδικό Id του κάθε χρήστη στην εφαρμογή και ένα συγκεκριμένο μυστικό κωδικό (το token αντιστοιχεί σε ένα JSON αντικείμενο όταν αποκρυπτογραφείται για να επικυρωθεί). Αυτό δημιουργείται κάθε φορά που κάνει login ο εκάστοτε χρήστης και γίνεται verify στην κάθε ενέργεια του χρήστη μέσα στην εφαρμογή, ώστε να γίνει η αυθεντικοποίηση/έλεγχος ταυτότητας και να επιβεβαιώνεται ότι ο χρήστης είναι αυτός που ισχυρίζεται πως είναι. Με αυτό τον τρόπο το token μεταφέρεται μεταξύ back και front-end μέσω των requests που γίνονται και προσθέτει ένα επιπλέον επίπεδο ασφάλειας στην λειτουργία της εφαρμογής. Με απλά λόγια, πρόκειται για ένα πακέτο που χρησιμοποιείται συχνά στο web programming ως κάποιο έξτρα authentication credential για τους χρήστες της εφαρμογής. Η εγκατάσταση του γίνεται με *npm install jwt-then* και η προσθήκη στο αρχείο js με *require('jwt-then')*.
- axios: Το axios αποτελεί άλλη μία γνωστή βιβλιοθήκη σχετικά με τον προγραμματισμό εφαρμογών ιστοτόπου/web programming στη γλώσσα javascript και εγκαθίσταται με την εντολή *npm install axios*. Χρησιμοποιείται στο front-end της εφαρμογής μας για να γίνουν http requests get και post, τα οποία πραγματοποιούνται στη μεριά του σέρβερ/back-end και συγκεκριμένα στον controller που έχουμε υλοποιήσει εκεί. Τα αιτήματα γίνονται με την μορφή promise, δηλαδή στέλνονται κάποια δεδομένα από το front στο back-end όπου συμβαίνουν διάφορες ενέργειες στη βάση δεδομένων π.χ. διάβασμα και έπειτα επιστρέφεται είτε μία απάντηση επιτυχώς στο front-end με τη μορφή *then(response => { ... })* (promise resolved) είτε κάποιο error που σημαίνει ότι το request απορρίφθηκε *catch(error => { ... })* (promise rejected). Γενικά τα promises είναι ιδιαίτερα σημαντικά για να γίνουν

ασύγχρονες διαδικασίες στην εφαρμογή μας, που σημαίνει ότι δε χρειάζεται να γίνει παύση της λειτουργίας της εφαρμογής ενώ πραγματοποιούνται τα promise requests και με αυτό τον τρόπο το αποτέλεσμα των promises επιστρέφεται όποτε είναι διαθέσιμο, χωρίς να υπάρχουν σφάλματα στις παρακάτω γραμμές κώδικα (λόγω του ότι π.χ. κάποια τιμή δεν υπάρχει επειδή δεν επιστράφηκε ακόμα απάντηση από το αίτημα).

- socket.io: Το library δημιουργεί κάποια session επικοινωνίας, δηλαδή one-time μοναδικές συνεδρίες μεταξύ του front και του back-end της εφαρμογής που χαρακτηρίζονται από κάποιο μοναδικό Id και βασίζεται στη σύνδεση websocket όπου υπάρχει αμφίδρομη επικοινωνία μεταξύ των 2 πλευρών, οι οποίες μπορούν και να στείλουν και να λάβουν δεδομένα. Έτσι, στην περίπτωση μας μέσω ενός socket instance θα στέλνονται τα μηνύματα που γράφει κάποιος χρήστης προς κάποιον άλλο από το front στο back-end και επίσης άλλες ευαίσθητες πληροφορίες όπως το id της τρέχουσας συνομιλίας και επιστρέφονται προς τον παραλήπτη χρήστη μέσω άλλου socket instance. Οι session συνδέσεις είναι ασφαλότερες, καθώς κάθε φορά υπάρχει διαφορετική σύνδεση με άλλα στοιχεία (διαφορετικό socket Id) και όταν γίνεται refresh/ανανέωση της ιστοσελίδας δημιουργείται αυτόματα άλλο socket session/instance. Τα sessions μπορούν να αναπτυχθούν με την εντολή *socket.on (...)* και εκτελούνται με την εντολή *socket.emit (...)* με την οποία στέλνονται τα δεδομένα που περιμένει να λάβει το *socket.on (...)*. Η λειτουργία του socket περιγράφεται καλύτερα μέσα στον κώδικα στο κεφάλαιο 4, ενώ η εγκατάσταση του γίνεται κατά τα γνωστά με *npm install socket.io*.
- tweetnacl/tweetnacl-util: Αναφέρεται στη βιβλιοθήκη που χρησιμοποιείται για να παραχθεί αρχικά το δημόσιο κλειδί του κάθε χρήστη που λειτουργεί σαν διεύθυνση για να σταλούν τα μηνύματα και σύμφωνα με αυτό το κλειδί μετά παράγεται το ζευγάρι δημοσίου-ιδιωτικού κλειδιού για τον αντίστοιχο χρήστη με το οποίο γίνεται α) η κρυπτογράφηση των μηνυμάτων που αποθηκεύονται στο BigchainDB

και στο τοπικό blockchain του MongoDB και β) η αποκρυπτογράφηση όταν θέλουμε να εμφανίσουμε τα μηνύματα στην συνομιλία για να τα διαβάσουν οι χρήστες (περισσότερες λεπτομέρειες στο κεφάλαιο 4). Η εγκατάσταση γίνεται πληκτρολογώντας *npm install tweetnacl tweetnacl-util*.

- crypto-js: Μία βιβλιοθήκη για την ενσωμάτωση κρυπτογραφικών αλγορίθμων μέσα στον javascript κώδικα μίας εφαρμογής. Συγκεκριμένα, εδώ χρησιμοποιείται μόνο ο αλγόριθμος sha512 της βιβλιοθήκης και πρόκειται για έναν αλγόριθμο hash, στον οποίο δίνεται μία είσοδος και προκύπτει η κρυπτογραφημένη έξοδος των 512 bits = 128 δεκαεξαδικών χαρακτήρων (κάθε δεκαεξαδικός χαρακτήρας αντιστοιχεί σε 4 bits). Δεν θα αναλυθεί πως λειτουργεί στο εσωτερικό του ο αλγόριθμος (αυτό ξεφεύγει από τους σκοπούς της εργασίας), ωστόσο ο αναγνώστης πρέπει να γνωρίζει πως κάθε φορά που μία είσοδος δίνεται για sha512 θα προκύπτει μία έξοδος με 128 δεκαεξαδικούς χαρακτήρες και αυτή είναι η κρυπτογραφημένη μορφή. Η ίδια είσοδος θα δίνει πάντα την ίδια έξοδο hash, όμως είναι σχεδόν αδύνατο να προκύψει από την κρυπτογραφημένη έξοδο η αρχική είσοδος. Στην παρούσα εργασία ο αλγόριθμος υλοποιείται για την κρυπτογράφηση των κωδικών των χρηστών της εφαρμογής και για τον υπολογισμό των hash των κουτιών του blockchain.
- cors: Το "CORS" σημαίνει Cross-Origin Resource Sharing και επιτρέπει να γίνονται requests από την εφαρμογή μας σε έναν άλλο ιστότοπο π.χ. από κάποια APIs που χρησιμοποιούμε. Ουσιαστικά, παρακάμπτει τις ρυθμίσεις που έχουν κάποιοι browsers και απαγορεύουν αυτά τα requests σύμφωνα με την πολιτική Same Origin Policy (SOP). Για να μην υπάρχουν θέματα σε διαφορετικούς browsers, το ενεργοποιούμε (θα δούμε στο κεφάλαιο 4 πως) στον κώδικα της εφαρμογής μας, αφού πρώτα το εγκαταστήσουμε με την εντολή *npm install cors*.

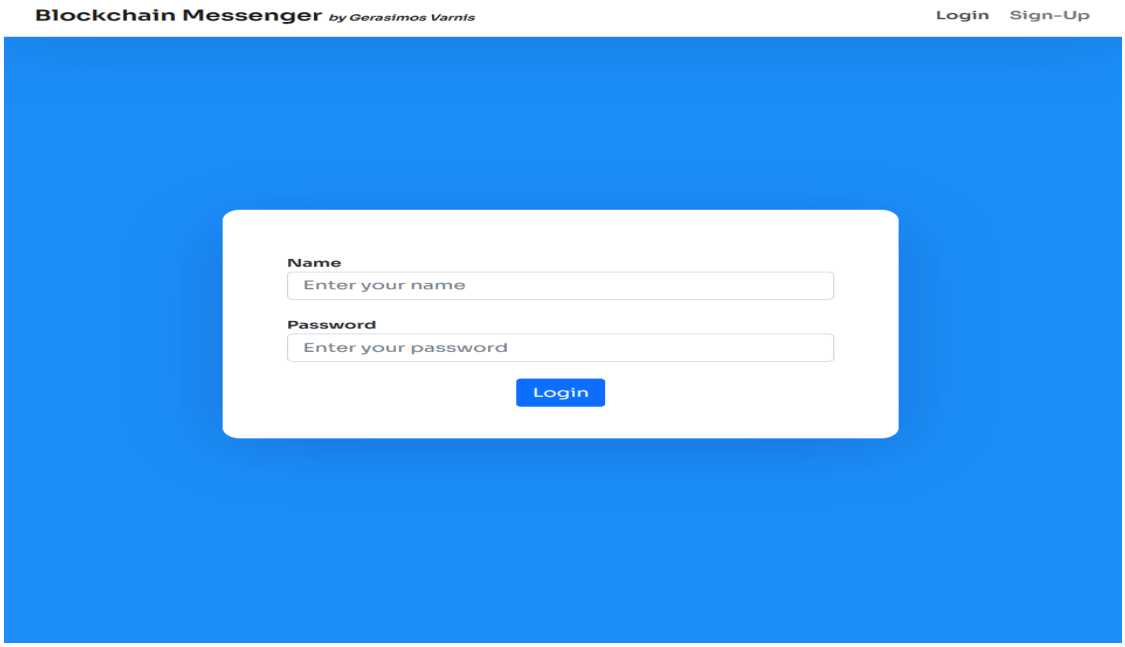
4. Λειτουργία εφαρμογής

Συνοπτικά, η λειτουργία της εφαρμογής μπορεί να χωριστεί σε 3 στάδια. Αρχικά, όταν κάποιος χρήστης ανοίγει την εφαρμογή συναντά την οθόνη Login, όπου μπορεί να πληκτρολογήσει τα στοιχεία του (username, password) και να εισέλθει στον λογαριασμό του ή να μεταβεί στην άλλη οθόνη του Sign-Up, να εγγραφεί στην εφαρμογή και έπειτα να πραγματοποιήσει την είσοδο του. Αν το login γίνει επιτυχώς ο χρήστης μεταβαίνει στο view του Lobby, όπου υπάρχει μία λίστα με όλους τους εγγεγραμένους χρήστες της εφαρμογής και μπορεί να επιλέξει με ποιόν επιθυμεί να συνομιλήσει και να κάνει 'enter' στη συζήτηση. Το 3ο και τελευταίο στάδιο είναι η συνομιλία/Conversation που περιλαμβάνει την live συνομιλία των 2 χρηστών και υπάρχουν όλα τα προηγούμενα μηνύματα που έχουν ανταλλάξει. Το κάθε μήνυμα είναι μία συναλλαγή στο blockchain του BigchainDB και κάθε τέτοια συναλλαγή είναι και ένα block στο blockchain που αποθηκεύεται στην τοπική μας βάση MongoDB. Υπάρχει μηχανισμός στον κώδικα μας, ώστε οποιαδήποτε μετατροπή/διαγραφή των κουτιών στο τοπικό μας blockchain αναιρείται με αυτόματη ανακατασκευή της αλυσίδας μας, 'τραβώντας' τα σχετικά δεδομένα από το blockchain του BigchainDB και αν κάτι τέτοιο συμβεί στη διάρκεια συνομιλίας 2 χρηστών, τότε ενημερώνονται με pop-up μήνυμα, περιμένουν 5 δευτερόλεπτα να επιστρέψει το blockchain στην κανονική του μορφή και μετά μπορούν να συνομιλήσουν ξανά.

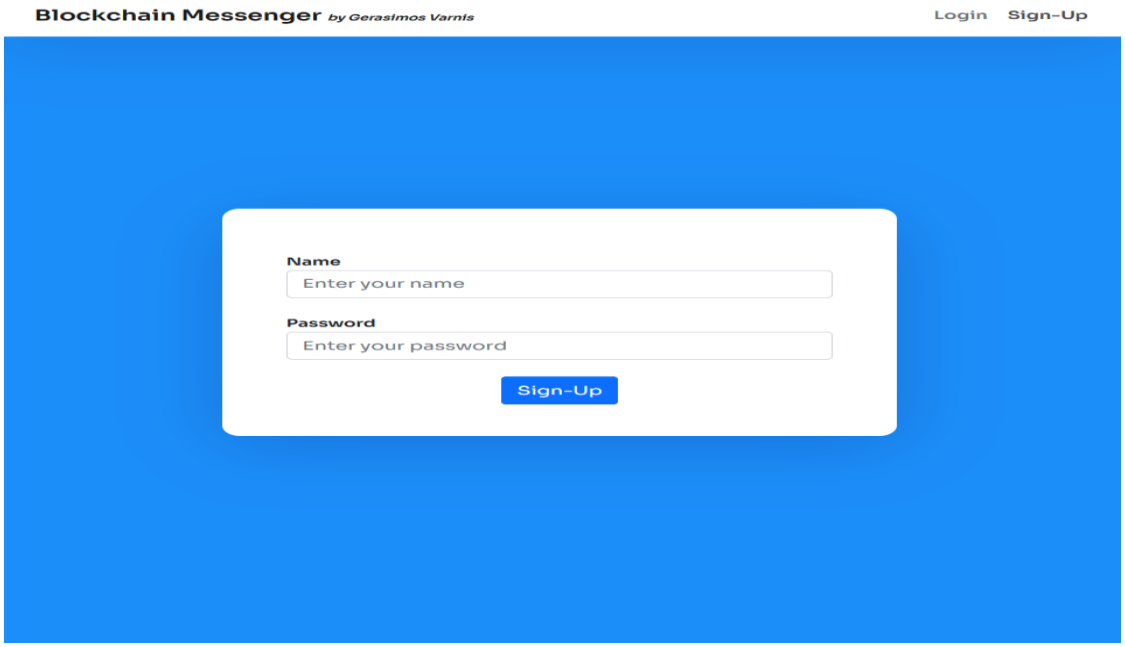
4.1 Login-SignUp

Αρχικά, έχουμε το αρχείο User.js (βρίσκεται στο back-end της εφαρμογής μας) που αναφέρεται στο μοντέλο του χρήστη (user model) και αντιστοιχεί στον πίνακα users που αποθηκεύεται στην τοπική μας βάση MongoDB. Υπάρχουν οι υποχρεωτικές ιδιότητες name, password και address_publicKey για τον κάθε χρήστη που μεταφράζονται στο όνομα, κωδικό πρόσβασης του στην εφαρμογή και στην διεύθυνση (ένα δημόσιο κλειδί) για την αποστολή μηνυμάτων προς τον συγκεκριμένο χρήστη, ενώ η ιδιότητα {timestamps:true} ενημερώνει τη βάση να βάζει αυτόματα σε κάθε εγγραφή την χρονική στιγμή που προστίθεται/αποθηκεύεται κάποιος χρήστης.

Οι 2 παρακάτω εικόνες 8 & 9 αντιστοιχούν στα views του Login και Sign-Up και αυτή είναι η οπτική του χρήστη, όταν ανοίγει την εφαρμογή και πρόκειται να εισέλθει ή να εγγραφεί σε αυτήν. Ο κώδικας που προσδιορίζει αυτά τα 2 pages χωρίζεται στα αρχεία του front-end (login.js, signup.js) και στα αρχεία του back-end (userController.js, userRouting.js, authentication.js). Επίσης, υπάρχουν αρχεία που εμπλέκονται έμμεσα στις διαδικασίες εισόδου ή εγγραφής, αλλά θα παρουσιαστούν την κατάλληλη στιγμή.



Εικόνα 8 - Login



Εικόνα 9 - Sign-Up

Ας περιγράψουμε τι συμβαίνει όταν ο χρήστης ανοίγει την εφαρμογή και θέλει να εγγραφεί:

- Σύμφωνα με την *Εικόνα 9* εισάγει τα στοιχεία που επιθυμεί (όνομα και κωδικό πρόσβασης) και πατάει το κουμπί Sign-Up ή το κουμπί Enter στο πληκτρολόγιο του.
- Τότε καλείται η μέθοδος signupUser(), λαμβάνονται τα στοιχεία που πληκτρολόγησε ο χρήστης από τα references των input elements και αυτές οι τιμές στέλνονται με το axios.post στη μέθοδο sign-up του back-end και συγκεκριμένα το request γίνεται στο url *http://localhost:8000/user/sign-up*.
- Στο αρχείο server.js έχει οριστεί πως για τα αιτήματα που γίνονται στο back-end θα χρησιμοποιείται η θύρα 8000 του localhost με την εντολή

```
const Server = app.listen(8000, () => {
    console.log('Server is listening on port 8000');
});
```

- Επίσης, στο αρχείο app.js υπάρχει η δήλωση πως μετά το *localhost:8000* θα βρίσκεται το */user*, οπότε τα αιτήματα θα πραγματοποιούνται μέσω του axios στο url *http://localhost:8000/user*.

```
app.use('/user', require('./userRouting'));
```

- Με βάση την τελευταία εντολή καλείται το αρχείο userRouting.js και εκεί ανάλογα με το τι υπάρχει στο url μετά το user/, που στην περίπτωση μας είναι το sign-up, καλείται η αντίστοιχη μέθοδος του userController από το αρχείο userController.js. Εκεί έχουμε την εντολή

```
router.post('/sign-up', appErrors(userController.signup));
```

Το router είναι μία λειτουργία του express και μεταφράζει τα αιτήματα που γίνονται από το front-end (μέσω του axios) σε διαδικασίες/μεθόδους στο back-end. Δηλαδή, ενώ ήμασταν στο *http://localhost:3000/signup* και κάναμε το axios post στο url *http://localhost:8000/user/sign-up* αυτό τελικά θα οδηγηθεί στον userController και στην μέθοδο signup και αν προκύψουν σφάλματα τότε θα διαχειριστούν από το αρχείο handlingErrors.js, καθώς όπως φαίνεται στο userRouting.js έχει γίνει και η δήλωση

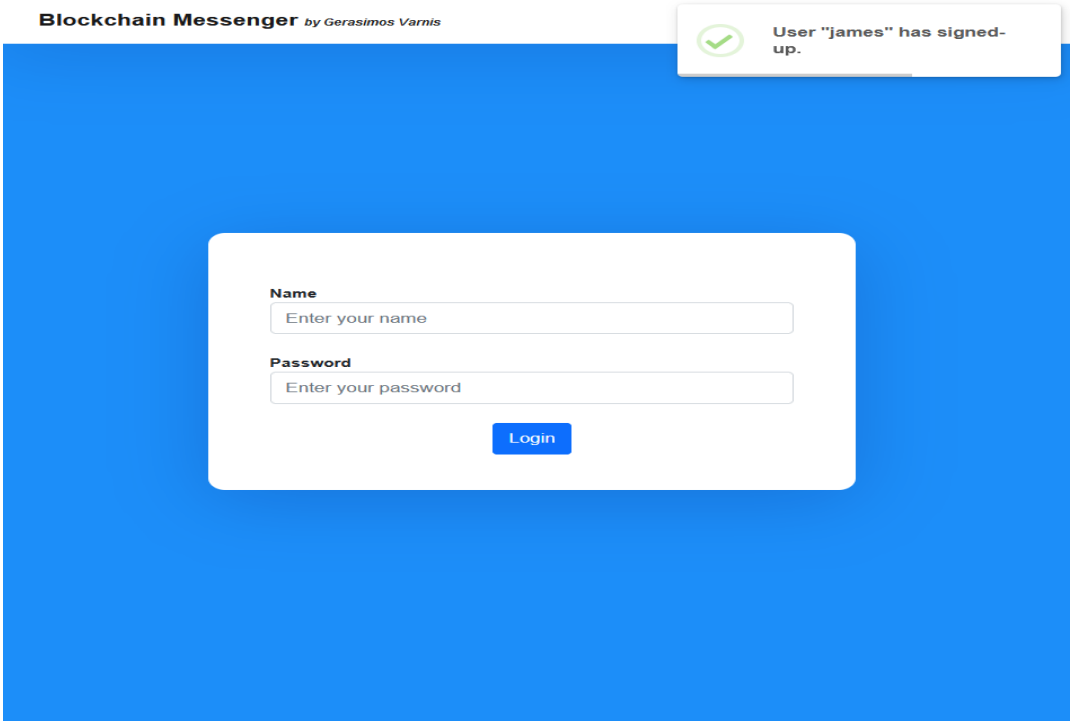
```
const {appErrors} = require('./handlingErrors')
```

- Τελικά, έχουμε φτάσει στο αρχείο userController.js και συγκεκριμένα στη μέθοδο signup. Παίρνουμε τα στοιχεία που εισήγαγε ο χρήστης μέσω του αντικειμένου req (request), ελέγχουμε στη βάση αν υπάρχει ήδη χρήστης με το όνομα ή αν ο κωδικός είναι μικρότερος από 8 χαρακτήρες ή αν δεν έχει εισαχθεί όνομα από τον χρήστη, ώστε να βγεί το αντίστοιχο error μήνυμα. Αν πάνε όλα καλά τότε παράγεται ένα δημόσιο κλειδί με βάση το εργαλείο nacl και το μετατρέπω σε δεκαεξαδική μορφή με τη μέθοδο *fromUint8ArraytoHexadecimalString*. Τελικά αποθηκεύω τον χρήστη στη βάση δεδομένων με το όνομα, το hash του κωδικού που πληκτρολόγησε και το δημόσιο κλειδί σε δεκαεξαδική μορφή και εντέλει επιστρέφω μία απάντηση json στο front-end μέσω του res.json ότι ο χρήστης έχει εγγραφεί επιτυχώς.
- Το αρχικό axios.post που είχε γίνει από το front-end στη μέθοδο signupUser() έχει πάρει πλέον πετυχημένη απάντηση ή κάποιο σφάλμα.

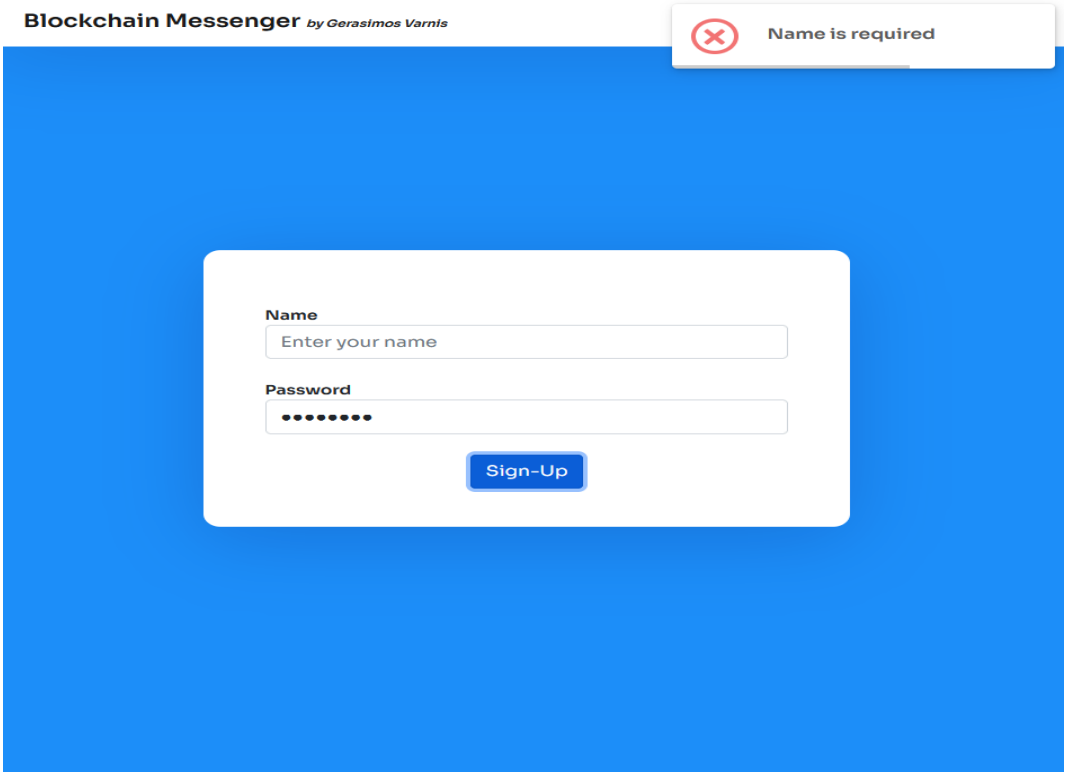
```
axios.post('http://localhost:8000/user/sign-up', {
            name,
            password
        }).then(response => {
            showMessage('success', response.data.message);
                 props.history.push('./login'); // αφού γίνει το πετυχημένο sign-up κατευθύνω τον χρηστη στο login για να εισάγει τα στοιχεία του και να κάνει είσοδο στην εφαρμογή. δηλαδή το push πρακτικά μετακινεί τον χρήστη από αυτό το view στο login
        }).catch(error => { // αν όμως υπήρξε error κατά το sign-up του χρήστη, μέσω του catch(error) παίρνουμε αυτό το σφάλμα και το εμφανίζουμε πάλι ανάλογα με το throw που έγινε μέσα στον userController.
            if(error.response){
                showMessage('error',error.response.data.message)
            }
        })
```

Η πετυχημένη απάντηση είναι μέσα στο then και εμφανίζεται στο χρήστη μέσω της showMessage και του pop up παραθύρου από το εργαλείο sweetalert (*Εικόνα 10*). Το σφάλμα 'πιάνεται' από την catch και αν πρόκειται για σφάλμα που ελέγχθηκε από τον controller τότε εμφανίζεται το μήνυμα που είχαμε ορίσει εκεί (*Εικόνα 11, Εικόνα 12,*

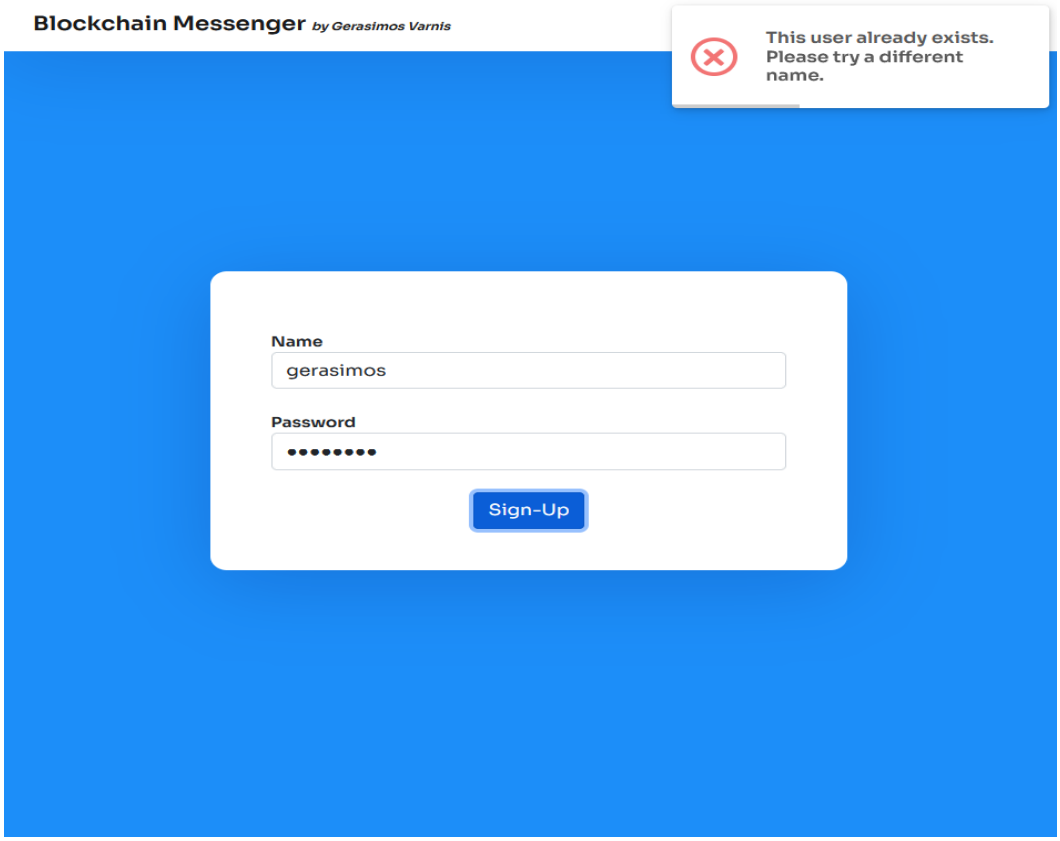
Εικόνα 13), αλλιώς εμφανίζεται το σφάλμα σύμφωνα με τις ρυθμίσεις του browser.



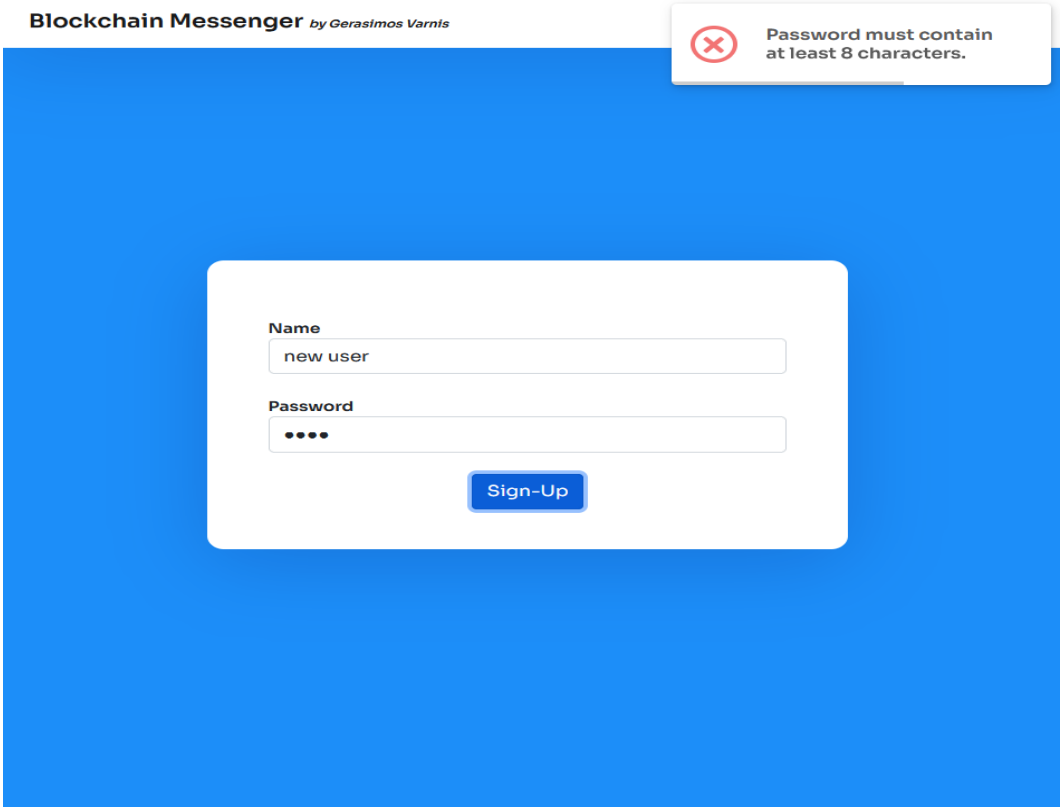
Εικόνα 10 - Επιτυχημένη εγγραφή και redirect στο login view



Εικόνα 11 - Σφάλμα κατά την εγγραφή -> Δεν έχει εισαχθεί όνομα



Εικόνα 12 - Σφάλμα κατά την εγγραφή -> Υπάρχει ήδη χρήστης με αυτό το όνομα



Εικόνα 13 - Σφάλμα κατά την εγγραφή -> Ο κωδικός πρόσβασης πρέπει να έχει τουλάχιστον 8 χαρακτήρες

Όταν ο χρήστης έχει πλέον εγγραφεί στην εφαρμογή, μπορεί να κάνει είσοδο και μάλιστα μετά την εγγραφή ανακατευθύνεται αυτόματα στο view του login, όπως φαίνεται και στην *Εικόνα 8*. Με παρόμοια λογική κώδικα όπως και στο sign-up, έχουμε:

- Σύμφωνα με την *Εικόνα 8* εισάγει τα στοιχεία του (όνομα και κωδικό πρόσβασης) και πατάει το κουμπί Login ή το κουμπί Enter στο πληκτρολόγιο του.
- Τότε καλείται η μέθοδος loginUser(), λαμβάνονται τα στοιχεία που πληκτρολόγησε ο χρήστης από τα references των input elements και αυτές οι τιμές στέλνονται με το axios.post στη μέθοδο login του back-end και συγκεκριμένα το request γίνεται στο url *http://localhost:8000/user/login*.
- Στο αρχείο server.js έχει οριστεί πως για τα αιτήματα που γίνονται στο back-end θα χρησιμοποιείται η θύρα 8000 του localhost με την εντολή

```
const Server = app.listen(8000, () => {
    console.log('Server is listening on port 8000');
});
```

- Επίσης, στο αρχείο app.js υπάρχει η δήλωση πως μετά το *localhost:8000* θα βρίσκεται το */user*, οπότε τα αιτήματα θα πραγματοποιούνται μέσω του axios στο url *http://localhost:8000/user*.

```
app.use('/user', require('./userRouting'));
```

- Με βάση την τελευταία εντολή καλείται το αρχείο userRouting.js και εκεί ανάλογα με το τι υπάρχει στο url μετά το user/, που στην περίπτωση μας είναι το login, καλείται η αντίστοιχη μέθοδος του userController από το αρχείο userController.js. Εκεί έχουμε την εντολή

```
router.post('/login, appErrors(userController.login));
```

Το router είναι μία λειτουργία του express και μεταφράζει τα αιτήματα που γίνονται από το front-end (μέσω του axios) σε διαδικασίες/μεθόδους στο back-end. Δηλαδή, ενώ ήμασταν στο *http://localhost:3000/login* και κάναμε το axios post στο url *http://localhost:8000/user/login* αυτό τελικά θα οδηγηθεί στον userController και στην μέθοδο login και αν προκύψουν σφάλματα

τότε θα διαχειριστούν από το αρχείο handlingErrors.js, καθώς όπως φαίνεται στο userRouting.js έχει γίνει και η δήλωση

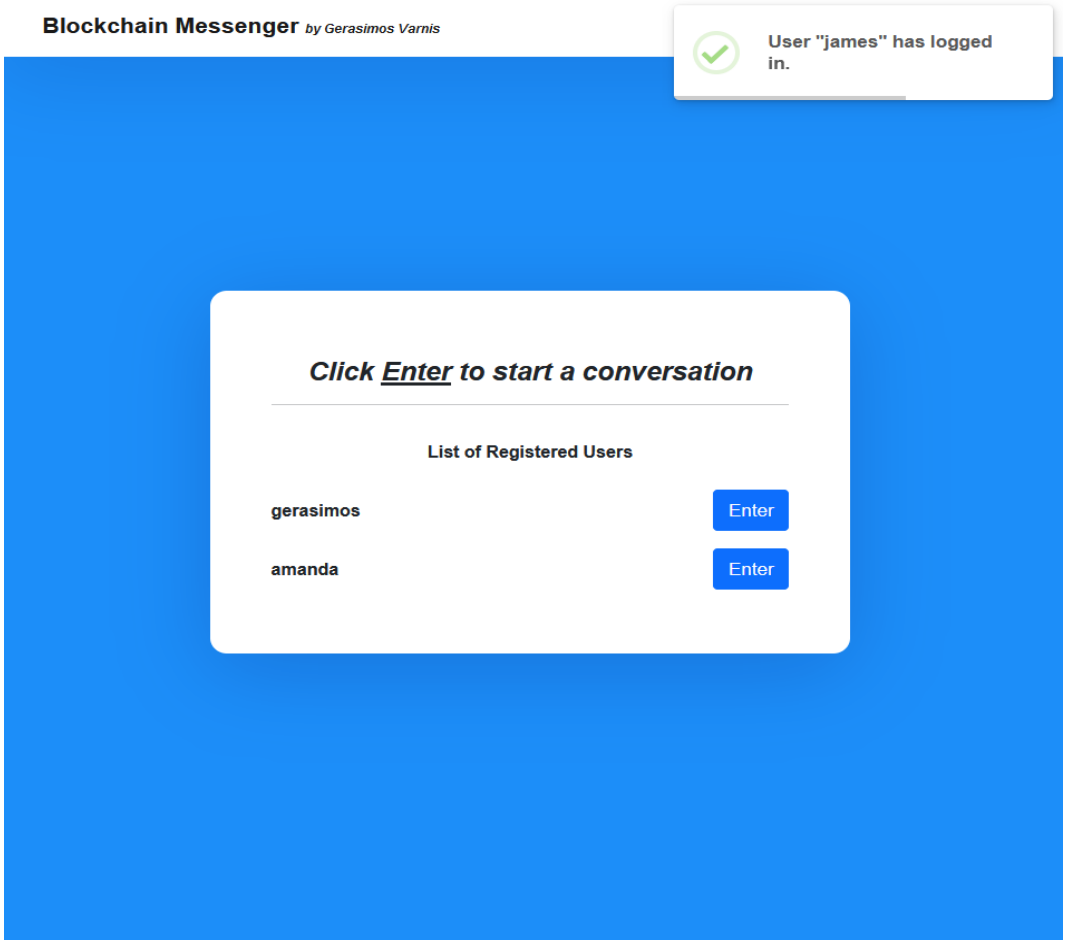
```
const {appErrors} = require('./handlingErrors')
```

- Τελικά, έχουμε φτάσει στο αρχείο userController.js και συγκεκριμένα στη μέθοδο login. Παίρνουμε τα στοιχεία που εισήγαγε ο χρήστης μέσω του αντικειμένου req (request), ελέγχουμε στη βάση αν υπάρχει χρήστης με αυτό το όνομα ή αν ο κωδικός είναι λανθασμένος, ώστε να βγεί το αντίστοιχο error μήνυμα. Αν πάνε όλα καλά τότε δημιουργείται το token με την μέθοδο sign του εργαλείου jwt, το οποίο είναι μοναδικό για αυτό τον χρήστη και τη συγκεκριμένη σύνδεση. Τελικά, επιστρέφω μία απάντηση json στο front-end μέσω του res.json ότι ο χρήστης έχει εισέλθει επιτυχώς με το αντίστοιχο token και το Id του από τη βάση δεδομένων.
- Το αρχικό axios.post που είχε γίνει από το front-end στη μέθοδο loginUser() έχει πάρει πλέον πετυχημένη απάντηση ή κάποιο σφάλμα.

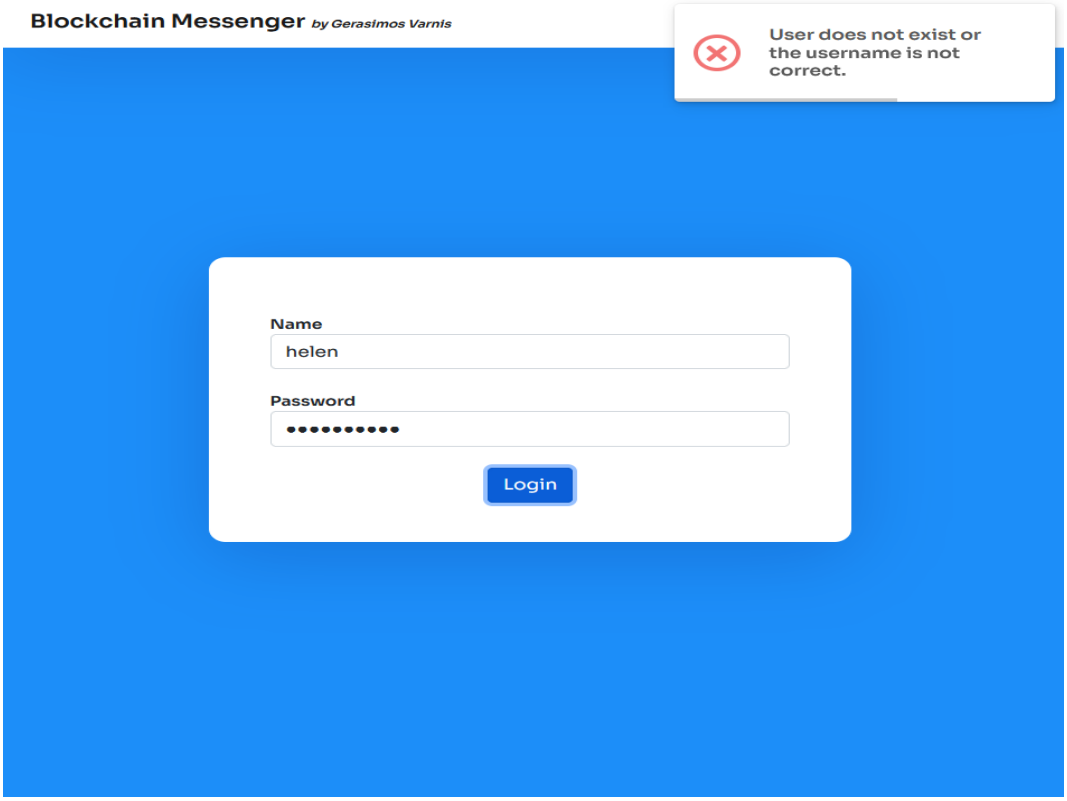
```
        axios.post('http://localhost:8000/user/login', {
            name,
            password
        }).then(response => {
            showMessage('success', response.data.message);
             // αποθηκεύω προσωρινά το token, το name και το Id του εκάστοτε χρηστη που κάνει login στο localStorage (token και Id υπάρχουν στο response.data), ώστε μετά να τα χρησιμοποιήσω σε άλλα views με το getItem
            localStorage.setItem('userToken', response.data.token);
            localStorage.setItem('userName', name);
        localStorage.setItem('currentUserId',response.data.userId);
          props.history.push('/lobby'); // αφού γίνει το πετυχημένο login κατευθύνω τον χρηστη στο lobby (όπου βλέπει όλους τους χρήστες της εφαρμογής και επιλέγει κάποιον για να συνομιλήσει). δηλαδή το push πρακτικά μετακινεί τον χρήστη από αυτό το view στο lobby
         props.SocketIO_Setup(); // αφού έγινε η πετυχημένη είσοδος στην εφαρμογή πρέπει να δώσουμε τιμή στο socket
        }).catch(error => { //αν όμως υπήρξε error κατά το login, μέσω του catch(error) παίρνουμε αυτό το σφάλμα και το εμφανίζουμε πάλι ανάλογα με το throw που έγινε μέσα στον userController.
            if(error.response){
                showMessage('error',error.response.data.message)
            }
```

```
        })
```

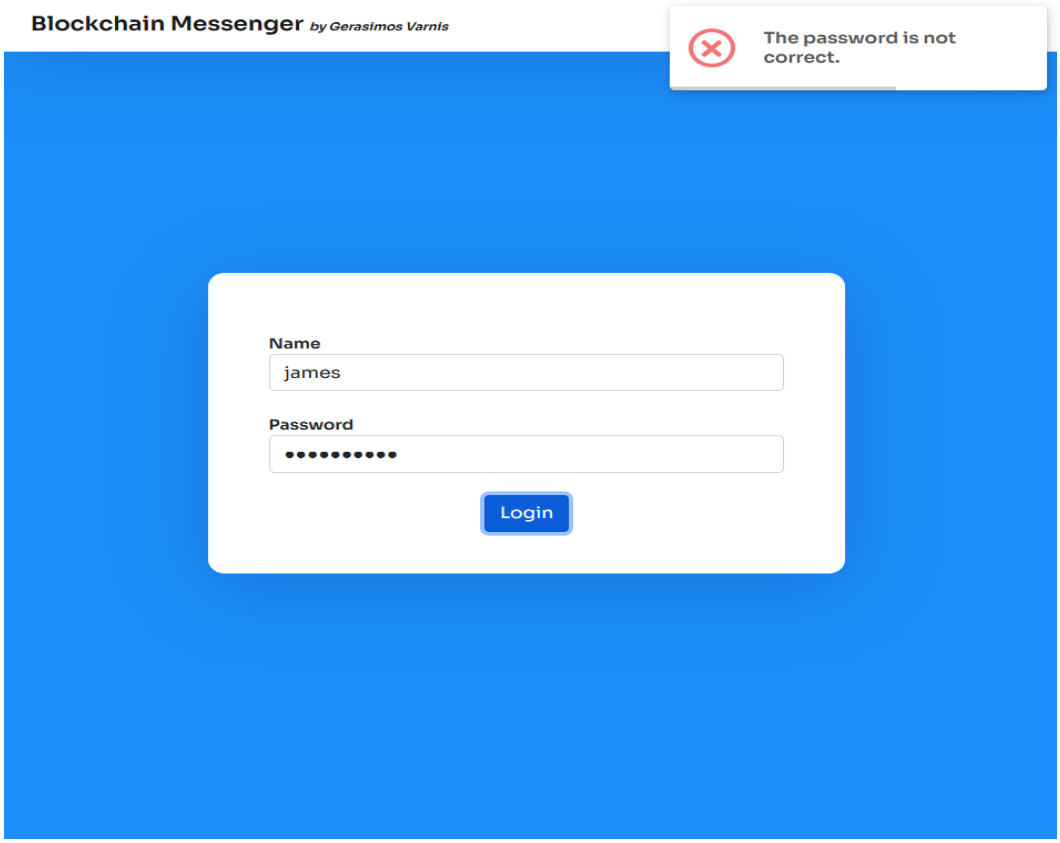
Η πετυχημένη απάντηση είναι μέσα στο then και εμφανίζεται στο χρήστη μέσω της showMessage και του pop up παραθύρου από το εργαλείο sweetalert (*Εικόνα 14*). Το σφάλμα 'πιάνεται' από την catch και αν πρόκειται για σφάλμα που ελέγχθηκε από τον controller τότε εμφανίζεται το μήνυμα που είχαμε ορίσει εκεί (*Εικόνα 15, Εικόνα 16*), αλλιώς εμφανίζεται το σφάλμα σύμφωνα με τις ρυθμίσεις του browser.



Εικόνα 14 - Επιτυχημένη είσοδος και lobby view



Εικόνα 15 - Σφάλμα κατά την είσοδο -> Το όνομα χρήστη δεν υπάρχει ή πληκτρολογήθηκε λάθος



Εικόνα 16 - Σφάλμα κατά την είσοδο -> Ο κωδικός πρόσβασης είναι λανθασμένος

4.2 Lobby

Μετά το πετυχημένο login του χρήστη μέσω της μεθόδου loginUser() του αρχείου login.js και συγκεκριμένα μετά τις παρακάτω γραμμές κώδικα,

```
props.history.push('/lobby'); // αφού γίνει το πετυχημένο login κατευθύνω τον χρηστη στο lobby (όπου βλέπει όλους τους χρήστες της εφαρμογής και επιλέγει κάποιον για να συνομιλήσει). δηλαδή το push πρακτικά μετακινεί τον χρήστη από αυτό το view στο lobby
props.SocketIO_Setup(); // αφού έγινε η πετυχημένη είσοδος στην εφαρμογή πρέπει να δώσουμε τιμή στο socket
```

μεταφερόμαστε στο view *http://localhost:3000/lobby* με παράλληλη κλήση της μεθόδου SocketIO_Setup(), η οποία είναι στο αρχείο App.js.

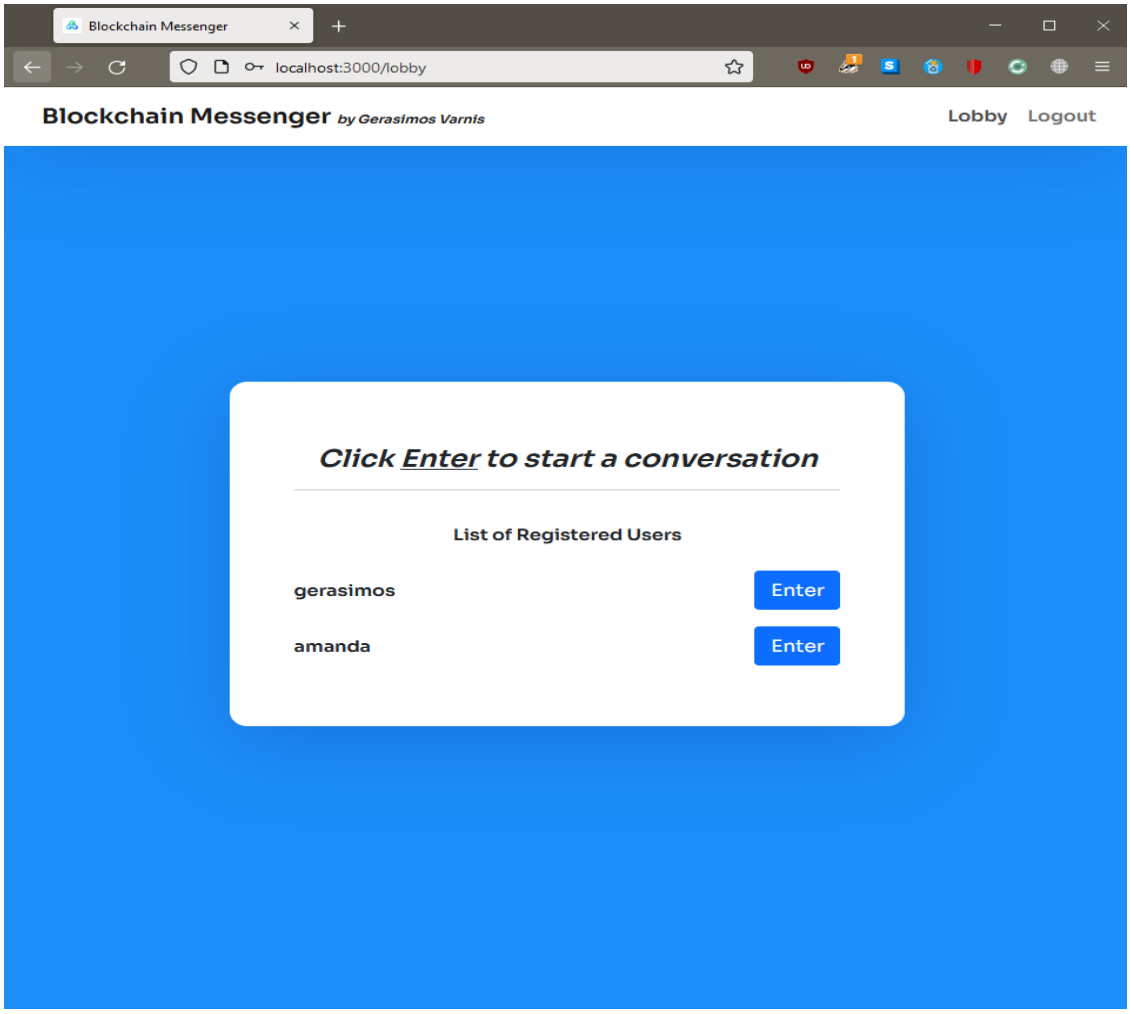
```
const SocketIO_Setup = () => {

    const userToken = localStorage.getItem('userToken');

      //αν υπάρχει το token χρήστη και δεν έχει πάρει ακόμα κανονική τιμή το SocketIO (είναι null δηλαδή και είτε δεν έχει γίνει ακόμα η σύνδεση είτε έγινε αποσύνδεση), τότε πρέπει να δημιουργήσω την σύνδεση με το socket ώστε να επικοινωνούμε με το backend
    if(userToken && !socketIO) {
      const socket = IO("http://localhost:8000", {
        query: {
            token: localStorage.getItem('userToken'),
        },
      });
      //όταν αποσυνδέεται η σύνδεση socket (πχ για λόγους αποσύνδεσης από το internet και όχι χειροκίνητα, δηλαδή με logout) πρέπει να θέσω το state του socket, το socketIO σε null
      socket.on('disconnect', () => {
        setSocketIO(null);
      });

      //όταν συνδέεται η σύνδεση socket πρέπει να θέσω το state του socket, το socketIO στην τιμή που πήρε από την επιτυχημένη σύνδεση που έγινε παραπάνω
      socket.on('connect', () => {
        setSocketIO(socket);
      });
    }
  }
```

Η μέθοδος αυτή αρχικά ελέγχει αν υπάρχει το token του χρήστη μέσω του localStorage (είχε αποθηκευτεί κατά το login) και ότι δεν υπάρχει ήδη σύνδεση socket ή δεν έχει γίνει αποσύνδεση. Τότε, δημιουργείται το socket instance δίνοντας σαν μοναδικό στοιχείο το token του χρήστη (που είναι επίσης μοναδικό) και προσδιορίζεται τί θα γίνεται στην αποσύνδεση του socket και στην σύνδεση. Στην αποσύνδεση το socket δεν έχει τιμή (θα είναι null), ενώ στη σύνδεση ορίζεται πως η διαδρομή για το socket και η επικοινωνία με το back-end θα

γίνεται μέσω του url *http://localhost:8000*, όπου θα υπάρχει το query σαν έξτρα πληροφορία στο request με το userToken να είναι το μοναδικό token του τρέχοντος χρήστη. Έτσι, η κάθε σύνδεση socket διαφέρει από την οποιαδήποτε άλλη και τελικά, με βάση τον κώδικα του App.js θα εμφανιστεί το Lobby component του αρχείου lobby.js στο οποίο έχει περαστεί σαν παράμετρος η σύνδεση socket που μόλις δημιουργήθηκε.

```
<Route exact path='/lobby' render={() => <Lobby socket={socketIO}/>}/>
```

Επομένως, βρισκόμαστε στο view της *Εικόνας 17* το οποίο αντιστοιχεί στο αρχείο lobby.js του front-end και παρατηρούμε ότι έχει εμφανιστεί μία λίστα με όλους τους διαθέσιμους χρήστες της εφαρμογής, με τους οποίους μπορεί να συνομιλήσει ο τρέχων χρήστης (james) πατώντας το ανάλογο κουμπί *Enter*.



Εικόνα 17 - Lobby page

Πως λειτουργεί το lobby:

- Αρχικά, ορίζουμε έναν πίνακα users με τη μορφή σταθεράς, που μπορεί να ενημερωθεί μόνο με την συνάρτηση setUsers(). Αυτή είναι η μορφή για μία λειτουργία του react που ονομάζεται React.useState([]) και ανήκει στην κατηγορία React Hooks, ένας τρόπος για να ενημερώνεται δυναμικά η κατάσταση ενός page στο react.
- Κάθε φορά που φορτώνει εκ νέου η σελίδα (γίνεται login, κάνω refresh ή πατάω το κουμπί *Lobby* πάνω δεξιά στη μπάρα/navbar) τρέχει η μέθοδος React.useEffect(...), που είναι άλλη μία σημαντική λειτουργία React Hooks.

```
React.useEffect(() => {
        getAllUsers();

      //αυτό είναι για την περίπτωση που γίνεται refresh στο
      conversation page. επειδή, η σύνδεση socket είναι session και
      χάνεται όταν γίνεται ανανέωση σελίδας. η συνομιλία λειτουργεί
      με τα συγκεκριμένα socket που έχει πάρει ο κάθε χρήστης (από
      τους 2 που συνομιλούν), δηλαδή αν ανανεώσω την σελίδα
      conversations δεν ισχύει πλέον το socket που είχα πριν και
      μου επέτρεπε να μιλήσω με τον άλλο χρήστη. για αυτό
      ανακατευθύνεται ο χρήστης εδώ όταν κάνει ένα τέτοιο refresh
      ώστε να ξαναπατήσει το enter. Ο σκοπός αυτού του μηνύματος
      είναι να ενημερωθεί ο χρήστης αν καταλάθος πατήσει το
      refresh/reload, ότι δεν υπάρχει λόγος να το ξανακάνει διότι η
      εφαρμογή λειτουργεί μια χαρά και realtime όπως και να χει.
      Επίσης, ο δεύτερος έλεγχος για το token γίνεται ώστε να μην
      βγαίνει αυτό το μήνυμα αν έχει αποσυνδεθεί ο χρήστης και πάει
      στο conversation και κάνει refresh.
        if(localStorage.getItem('reloaded')==='true'
          && localStorage.getItem('userToken') !== 'null'){
           showMessage('info','You were redirected here to
                       instantiate the socket connection again.
                       There is no need to refresh the page.');
           localStorage.setItem('reloaded', false);
        }

         //μήνυμα ενημέρωσης του χρήστη για την περίπτωση που το
          τοπικό blockchain μας δεν είναι έγκυρο όταν ο χρήστης
          κάνει enter στη συζήτηση. αυτό εδώ είναι το τελευταίο
          βήμα. ξεκινάμε από το backend, πάμε στο frontend στο
          αρχείο conversation.js και από εκεί καταλήγουμε εδώ.
        if(localStorage.getItem('reconstructedWhenEntering')
          === 'true'){
         showMessage('info','The blockchain is not valid and has
                       to be reconstructed. Please wait for this
                       message to disappear and enter again...');
          localStorage.setItem('reconstructedWhenEntering', false);
        }
    }, []);
```

- Μέσα στην useEffect() εκτός από κάποιους ελέγχους και τα σχετικά μηνύματα, για το αν το blockchain μας δεν είναι έγκυρο ή αν γίνει ανανέωση στο view της συνομιλίας (όπως θα δούμε, το τρέχον socket αποσυνδέεται αν γίνει ανανέωση της σελίδας), υπάρχει και η κλήση της μεθόδου getAllUsers() που κάνει ένα axios get για να φέρει τους χρήστες από την βάση δεδομένων και να τους εμφανίσει στην λίστα. Σύμφωνα με τον κώδικα που ακολουθεί, στο εσωτερικό της getAllUsers() υπάρχει αυτή τη φορά μία get μέθοδος που αντιστοιχεί σε αίτημα στο url *http://localhost:8000/user/users* και τα δεδομένα που στέλνονται στο request είναι ένα αντικείμενο headers, το οποίο περιλαμβάνει την ιδιότητα *Authorization: 'Bearer ' + localStorage.getItem('userToken')*. Όπως αναφέρεται και στον σχολιασμό του κώδικα, ο τρόπος που στέλνουμε το get request είναι μέσω πιστοποίησης του χρήστη, ήτοι το bearer authentication ή αλλιώς token authentication σημαίνει "δώσε πρόσβαση για request από τον client προς τον server, μόνο σε αυτόν που φέρει το token (bearer of token)" και αποτελεί έναν τρόπο να γίνει ένα http request προς τον σέρβερ με έλεγχο ταυτότητας/authentication. Έπειτα, το token αυτό ελέγχεται στο backend/server side και αν είναι όντως έγκυρο, τότε γίνεται το request κανονικά.

```
// αυτή η μέθοδος είναι μέσα στο useEffect που έχει σαν παράμετρο το []. αυτό σημαίνει πως ότι βρίσκεται μέσα στο useEffect θα κληθεί μόνο την πρώτη φορά που γίνεται render το component Lobby. Επομένως, όταν ο χρήστης κάνει πετυχημένο login και πάει σε αυτό το view, τότε καλείται αυτή η μέθοδος για να φέρει όλους τους χρήστες από την βάση και να τους εμφανίσει. Γίνεται ένα get request μέσω του axios.get προς τον σέρβερ μας. Σε αυτό το request θέτουμε κάποιους headers και υπάρχει η ιδιότητα του Authentication. Αυτό γίνεται, επειδή χρησιμοποιούμε τα jwt - JSON Web Tokens για να πιστοποιήσουμε τον χρήστη όταν κάνει το login μέσω ενός token που υπογράφεται και ελέγχεται, σύμφωνα με το μοναδικό Id του κάθε χρήστη. Ο τρόπος λοιπόν που στέλνουμε το get request εδώ, είναι μέσω πιστοποίησης του χρήστη. δηλαδή το bearer authentication ή αλλιώς token authentication σημαίνει "δώσε πρόσβαση για request από τον client προς τον server, μόνο σε αυτόν που φέρει το token (bearer of token)". Έπειτα, το token αυτό ελέγχεται στο backend/server side και αν είναι όντως έγκυρο, τότε γίνεται το request.
    const getAllUsers = () => {
        axios.get('http://localhost:8000/user/users', {
            headers: {
```

```
                Authorization: 'Bearer ' +
                                localStorage.getItem('userToken'),
            },
        })
        .then( (response) => {
              setUsers(response.data);  // λοιπόν εδώ με την get παίρνω όλα τους χρήστες που αντιστοιχούν ο καθένας σε μία συνομιλία για τον χρηστη με αυτό το συγκεκριμένο token, άρα το response.data θα μου επιστρέφει έναν πινακα με όλους τους χρήστες (που αργότερα μεταφράζονται σε συνομιλίες/conversations. Αν επιλέξω να κάνω enter σε έναν χρήστη από την λίστα, τότε μπαίνω στο conversation με αυτό τον χρήστη). Έτσι, έχοντας μπεί στο then υπάρχει το αποτέλεσμα της επιτυχημένης get και επομένως θέτω το state μου users στο αποτέλεσμα αυτό.
        })
        .catch( (error) => {
             setTimeout(getAllUsers, 2000); // αν υπάρχει error δεν εμφανίζουμε κάποιο μήνυμα αλλά προσπαθούμε να ξαναφέρουμε όλους τους χρήστες μετά από 2 δευτερόλεπτα (πχ ίσως υπήρχε κάποια αδυναμία σύνδεσης με το backend)
        });
    };
```

- Στο αρχείο server.js έχει οριστεί πως για τα αιτήματα που γίνονται στο back-end θα χρησιμοποιείται η θύρα 8000 του localhost με την εντολή

```
const Server = app.listen(8000, () => {
    console.log('Server is listening on port 8000');
});
```

- Επίσης, στο αρχείο app.js υπάρχει η δήλωση πως μετά το *localhost:8000* θα βρίσκεται το */user*, οπότε τα αιτήματα θα πραγματοποιούνται μέσω του axios στο url *http://localhost:8000/user*.

```
app.use('/user', require('./userRouting'));
```

- Με βάση την τελευταία εντολή καλείται το αρχείο userRouting.js και εκεί ανάλογα με το τι υπάρχει στο url μετά το user/, που στην περίπτωση μας είναι το users, καλείται η αντίστοιχη μέθοδος του userController από το αρχείο userController.js. Εκεί έχουμε την εντολή

```
router.get('/users',
authentication,appErrors(userController.getAllUsers));
```

Το router είναι μία λειτουργία του express και μεταφράζει τα αιτήματα που γίνονται από το front-end (μέσω του axios) σε διαδικασίες/μεθόδους στο back-end. Δηλαδή, ενώ ήμασταν στο *http://localhost:3000/lobby* και κάναμε το axios get στο url *http://localhost:8000/user/users,* αυτό τελικά θα οδηγηθεί στον userController και στην μέθοδο getAllUsers() και αν προκύψουν

σφάλματα τότε θα διαχειριστούν από το αρχείο handlingErrors.js, καθώς όπως φαίνεται στο userRouting.js έχει γίνει και η δήλωση

```
const {appErrors} = require('./handlingErrors')
const authentication = require('./authentication');
```

Το authentication μέσα στην παραπάνω εντολή router, πραγματοποιεί τον έλεγχο που περιγράψαμε και αυτό γίνεται σύμφωνα με το αρχείο authentication.js. Εκεί παίρνουμε το token από το header του αιτήματος που έγινε αρχικά με την axios get και ελέγχουμε αν υπάρχει token (ώστε να βγεί μήνυμα λάθους και να μην επιστραφεί πετυχημένη απάντηση). Αν το token υπάρχει, πρέπει να επικυρωθεί με την μέθοδο verify() του jwt και αν προκύψει οποιοδήποτε σφάλμα στην επικύρωση θα 'πιαστεί' από την catch και θα εμφανιστεί μήνυμα 'Not Authorized'. Αν το token είναι εντέλει έγκυρο με την next() μέθοδο του express framework, επιστρέφεται αυτόματα το request στην μέθοδο getAllUsers().

- Τελικά, έχουμε φτάσει στο αρχείο userController.js και συγκεκριμένα στη μέθοδο getAllUsers(). Βρίσκουμε όλους τους χρήστες από την συλλογή User στη βάση δεδομένων και τους επιστρέφουμε ως απάντηση json στο front-end μέσω του res.json.
- Το αρχικό axios.get που είχε γίνει από το front-end στη μέθοδο getAllUsers(), έχει πάρει πλέον πετυχημένη απάντηση ή κάποιο σφάλμα. Η πετυχημένη απάντηση είναι μέσα στο then και ενημερώνεται ο πίνακας users. Το σφάλμα 'πιάνεται' από την catch και θέτουμε να ξαναγίνει το axios.get σε 2 δευτερόλεπτα μέχρι να γίνει πετυχημένη επιστροφή απάντησης της λίστας των χρηστών.
- Εφόσον έχουμε τη λίστα με τους χρήστες, αυτή εμφανίζεται μέσω ενός filter και map, μέσα στο * ... * της *Εικόνας 18*. Οι χρήστες που έχουμε πάρει από την axios.get ως απάντηση φιλτράρονται πρώτα, ώστε να αφαιρεθεί ο τρέχων χρήστης της εφαρμογής (άρα θα εμφανιστούν όλοι οι υπόλοιποι χρήστες) και για τον νέο πίνακα (χωρίς τον τρέχων χρήστη) γίνεται ένα map, που σημαίνει για κάθε στοιχείο του πίνακα (για κάθε χρήστη) θα εμφανίσω

2 divs και ένα Link. Το πρώτο div ομαδοποιεί τα στοιχεία του εκάστοτε χρήστη σύμφωνα με το μοναδικό του user._id και μέσα σε αυτό υπάρχει το div με το όνομα του και ένα Link, που όταν πατήσω το κουμπί *Enter* με μεταφέρει για παράδειγμα στο url http://localhost:3000/conversation/6258797276236498_61965f50679cab98f4e7cf39_gerasimos . Μετά το /conversation/ παίρνω α) μόνο τους αριθμούς από τα ids των 2 χρηστών (μέσω του regex *const reg = /\d+/g* και *match(reg).join('')*) που πρόκειται να συνομιλήσουν και τους προσθέτω (αυτός ο αριθμός θα είναι πάντα ο ίδιος), β) το id αυτού που θα λαμβάνει τα μηνύματα και γ) το όνομα του. Τα 3 αυτά στοιχεία χωρίζονται με _ και θα καθιστούν μοναδική την κάθε συνομιλία μεταξύ 2 χρηστών.

```
<div className='conversations'>
    <div className='text'>List of Registered Users<br /><br />
        {users.filter(u => u.name !== localStorage.getItem('userName')).map(user => (
            <div key={user._id} className="conversation">
                <div>{user.name}</div>
                {/* περνάμε σαν παράμετρο στο url της κάθε συνομιλίας, α) το άθροισμα
                των αριθμών που υπάρχουν στα ids των 2 χρηστών που συνομιλούν ( το
                οποίο θα είναι πάντα σταθερό), β) το id αυτού που θα λαμβάνει τα
                μηνύματα και γ) το όνομα του. αυτά είναι αναγνωριστικά που βοηθάνε
                στην ανταλλαγή των μηνυμάτων με σωστό τρόπο μετά στο conversation page
                */ }
                <Link to={'/conversation/' + (parseInt(user._id.match(reg).join('')) +
                parseInt(localStorage.getItem('currentUserId').match(reg).join(''))) +
                '_' + user._id + '_' + user.name}>
                    <button className="btn btn-primary btn-block">Enter</button>
                </Link>
            </div>
        ))}
    </div>
</div>
```

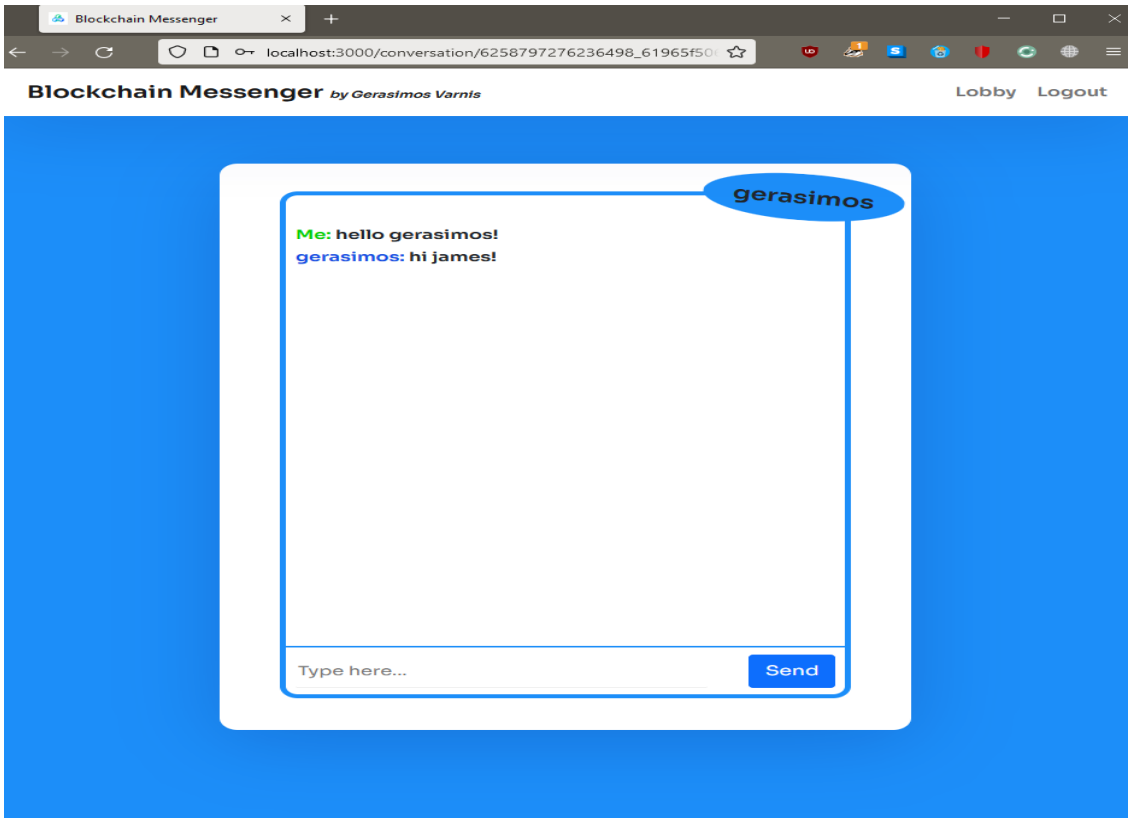
Εικόνα 18 - Πως προκύπτει η λίστα των χρηστών

4.3 Conversation

Εφόσον ο χρήστης επιλέξει κάποιον άλλο χρήστη για να συνομιλήσει και πατήσει το αντίστοιχο κουμπί *Enter* στην *Εικόνα 17*, οι παράμετροι μετά το /conversation/ της *Εικόνας 18* (που εξηγήσαμε παραπάνω) στέλνονται στο αρχείο App.js και αποτελούν το conversationParameters του παρακάτω κώδικα.

```
{/* με την : μετά url περνάω ότι είναι μετά την : σαν params από αυτό το
page στο conversation και έτσι όπως θα δούμε σε εκείνο το view μπορώ να πάρω
την τιμή των παραμέτρων αυτών μέσω του match.params. πρακτικά αυτά εδώ θα
είναι αυτά που περνάνε από το lobby, όταν πατάω το κουμπί enter σε κάποια
συνομιλία */}
<Route exact path='/conversation/:conversationParameters' render={(props) =>
      <Conversation socket={socketIO} match={props.match}/>}
/>
```

Έτσι, θα μεταφερθούμε στο Conversation component όπου θα περάσουμε ότι περιέχεται στο conversationParameters στο props.match ως match και την τιμή της σταθεράς socketIO (που είχαμε ορίσει μέσω της μεθόδου SocketIO_Setup() όταν έγινε το login και η επεξήγηση βρίσκεται στην αρχή του υποκεφαλαίου 4.2, μερικές σελίδες νωρίτερα) ως socket. Τελικά, θα βρεθούμε στο conversation view το οποίο φαίνεται στην *Εικόνα 19*.



Εικόνα 19 - Παράδειγμα συνομιλίας - conversation page

Ακολουθεί η επεξήγηση της λειτουργίας του conversation:

- Αρχικά, παίρνουμε όλες τις πληροφορίες που περιέχονται στο match.params.conversationParameters και συγκεκριμένα θα πάρω το id της συνομιλίας, το id του χρήστη που λαμβάνει τα μηνύματα και το όνομα του. Ορίζουμε έναν πίνακα messages με τη μορφή σταθεράς, που μπορεί να ενημερωθεί μόνο με την συνάρτηση setMessages() και αναφέρεται σε όλα τα μηνύματα που έχουν ανταλλάξει οι 2 χρήστες της συνομιλίας. Αυτή είναι η μορφή για μία λειτουργία του react που ονομάζεται React.useState([]) και ανήκει στην κατηγορία React Hooks, ένας τρόπος για να ενημερώνεται δυναμικά η κατάσταση ενός page στο react.

```
const conversationID  =
match.params.conversationParameters.split('_')[0];
// αυτό είναι το άθροισμα των αριθμών που υπάρχουν στο id του κάθε
εκ των 2 χρηστών. θα είναι πάντα ίδιο είτε μπει ο ένας ως
αποστολέας είτε ο άλλος. άρα, σύμφωνα με αυτό θα στέλνονται τα
μηνύματα και είναι το μοναδικό αναγνωριστικό της κάθε συνομιλίας
const userID = match.params.conversationParameters.split('_')[1];
// id του χρήστη που λαμβάνει τα μηνύματα
var receiverName =
match.params.conversationParameters.split('_')[2]; // όνομα του
χρήστη που λαμβάνει τα μηνύματα και μπαίνει ως τίτλος στη συνομιλία
πάνω δεξιά. δηλαδή όταν μπαίνω στην συνομιλία ξέρω ότι αυτός που
συνομιλώ είναι με το πάνω δεξιά όνομα

const [messages, setMessages] = React.useState([]); // το state των
μηνυμάτων που υπάρχουν στην συνομιλία

const msgReference = React.useRef(); // δημιουργώ το reference,
ώστε να υπάρχει δυναμική αναφορά σε σχέση με το πεδίο του μηνύματος
που γράφει ο χρήστης και έτσι έχω άμεση πρόσβαση με το
current.value
```

- Υπάρχουν 2 συναρτήσεις React.useEffect(...), που είναι άλλη μία σημαντική λειτουργία React Hooks και αυτό γίνεται επειδή θέλουμε κάθε useEffect() να δουλεύει με διαφορετικό τρόπο. Η πρώτη useEffect() θα φορτώνει μόνο την πρώτη φορά που εισέρχεται κάποιος χρήστης στη συνομιλία και φαίνεται παρακάτω, όπου το emit καλεί μία μέθοδο socket και το on ορίζει τι θα γίνεται όταν καλείται η αντίστοιχη μέθοδος.

```
//μόνο την πρώτη φορά που φορτώνει η σελίδα (όταν μπαίνει ο χρήστης
πρώτη φορά μετά το πάτημα κουμπιού enter)
    React.useEffect(() => {
```

```
        // αν υπάρχει η σύνδεση socket συνδέεται ο χρήστης στην
συζήτηση, καλώντας την μέθοδο enterConversation
        console.log(socket);
        if(socket){
            socket.emit('enterConversation', {
                conversationId: conversationID,
                userId: userID
            });

            //αν όταν κάνουμε enter σε μία συζήτηση, εξακριβωθεί
από τους ελέγχους στο backend ότι το blockchain στην βάση μας δεν
είναι έγκυρο, τότε γίνεται ανακατεύθυνση στο lobby όπου εμφανίζεται
μήνυμα ότι ανακατασκευάζεται το blockchain και πρέπει να γίνει
enter ξανά.
           socket.on('informTheUserWhenEntering', () => {
             window.location.href = 'http://localhost:3000/lobby';
             localStorage.setItem('reconstructedWhenEntering',
                 true);
            })

            //φέρνει το ιστορικό όλων των μηνυμάτων από το
blockchain της βάσης
            socket.on('getAllMessages', ({allMessages}) => {
                var newMessages = [];
                if(allMessages){
                    for (const msg of allMessages) {
                        const message = {
                            name: msg.username,
                            userId: msg.userSender,
                            message: msg.message
                        }
                        newMessages.push(message);
                    }
                    setMessages(newMessages);
                }
            });
        }

        // όταν φεύγει ο χρήστης από την συνομιλία (κλείνει
browser, πάει πίσω, κάνει refresh), τότε καλούμε την μέθοδο
leaveConversation
        return () => {
           if(socket){
                 socket.emit('leaveConversation', {
                        conversationId: conversationID,
                 })
           }
        }
},[]);
```

- Επομένως, αρχικά καλείται η μέθοδος enterConversation που υπάρχει στο αρχείο server.js του back-end και εκεί στέλνονται οι παράμετροι που περνάμε (conversationID, userID).

```
  socket.on('enterConversation', async ({conversationId, userId})
    => {
     socket.join(conversationId);
```

```
    console.log('entered the conversation ' + conversationId)

    const blockchain = await Blockchain.find(); // επιστρέφει όλο
το blockchain

    const res = await conn.searchAssets('BlockchainMessenger_v1');

        // αν δεν υπάρχει το blockchain και επίσης δεν υπάρχει και
στα αρχεία του bigchaindb, τότε σημαίνει πως είναι η πρώτη φορά που
γίνεται εισαγωγή στον πίνακα αυτόν στην βάση, άρα πρέπει να
δημιουργήσω το genesis block. Αυτό θα γίνει μόνο αν ισχύουν τα
παραπάνω και επίσης το μήκος του πίνακα από το bigchaindb είναι 0,
δηλαδή δεν υπάρχει τέτοιο blockchain με αυτό το όνομα
        if(blockchain.length === 0 && res.length === 0) {
            await CreateGenesisBlock(socket.userId);
        }
        else{
            const valid = await VerifyTheBlockchain(blockchain);
            if(!valid){
                await ReconstructTheBlockchain();
                SocketIO.to(conversationId).emit
                    ('informTheUserWhenEntering', {});
            }
        }

        const userS = await User.findOne({_id: socket.userId}); //
          αποστολέας
        const userR = await User.findOne({_id: userId}); //
          παραλήπτης

        const senderKeyPair = nacl.box.keyPair.fromSecretKey(
    fromHexadecimalStringtoUint8Array(userS.address_publicKey));
        const recipientKeyPair = nacl.box.keyPair.fromSecretKey(
    fromHexadecimalStringtoUint8Array(userR.address_publicKey));

        const oneTimeCode1 = recipientKeyPair.publicKey.slice(8);
        const oneTimeCode2 = senderKeyPair.publicKey.slice(8);


        // βρίσκω τα blocks που περιέχουν τα μηνύματα μεταξύ των 2
χρηστών είτε αυτά που χει στείλει ο ένας και τα παρέλαβε ο άλλος
είτε το αντίστροφο
        const blocks = await Blockchain.find({$or:[ {senderId:
          socket.userId, recipientId: userId}, {senderId: userId,
          recipientId: socket.userId} ] })
        const userMessages = [];
        if(blocks.length>=1){
            for(var block of blocks){
                let decodedMessage =
                 block.senderId === socket.userId?
                        nacl.box.open(
    fromHexadecimalStringtoUint8Array(block.message), oneTimeCode1,
    senderKeyPair.publicKey, recipientKeyPair.secretKey) :
                        nacl.box.open(
    fromHexadecimalStringtoUint8Array(block.message), oneTimeCode2,
    recipientKeyPair.publicKey, senderKeyPair.secretKey);
```

```
                let readableMessage =
                        nacl.util.encodeUTF8(decodedMessage);

                const message = {
                    message: readableMessage,
                    username: block.senderId === socket.userId ?
                        userS.name : userR.name,
                        userSender: block.senderId
                }
                userMessages.push(message);
            }

            if(userMessages && userS && userR){
                SocketIO.to(conversationId).emit('getAllMessages',
{
                    allMessages: userMessages
                });
            }
        }
    });
```

Εδώ γίνονται διάφορες διαδικασίες που είναι σημαντικές για την εφαρμογή μας α) επιστρέφεται όλο το blockchain της εφαρμογής μας από τη τοπική βάση μας, β) βρίσκω όλες τις συναλλαγές που αντιστοιχούν στο όνομα 'BlockchainMessenger_v1' (που το δίνω σαν όνομα σε κάθε συναλλαγή/μήνυμα που αποθηκεύω στο BigchainDB), γ) ελέγχω αν υπάρχει το blockchain μας με το συγκεκριμένο όνομα (αν δεν υπάρχει το blockchain στη τοπική βάση και επίσης δεν υπάρχει και στα αρχεία του bigchaindb, τότε σημαίνει πως είναι η πρώτη φορά που θα δημιουργηθεί η συλλογή στην βάση, άρα πρέπει να δημιουργήσω το genesis block, το πρώτο κουτί του blockchain που πρέπει να μπει χειροκίνητα, με την μέθοδο createGenesisBlock(...)),

```
// αυτή θα καλείται μια φορά όταν πχ δημιουργώ πρώτη φορα το
blockchain μου με το συγκεκριμένο όνομα (που σημαίνει ότι στη βάση
δεν υπάρχει τίποτα στον πινακα blockchain), μετά θα υπάρχει μόνιμα
όλο το blockchain μας στη bigchaindb και όποτε εντοπίζεται άκυρο
blockchain τότε θα το φέρνω από εκεί και θα γίνεται reconstruct με
την αντίστοιχη μέθοδο
const CreateGenesisBlock = async(idS) => {
    const userS = await User.findOne({_id: idS}); // βρίσκω τον
                χρήστη που ενεργεί ως αποστολέας
    //δημιουργώ με βάση το δημόσιο κλειδί που έχει στην τοπική
βάση, ένα ζευγάρι κλειδιά Ed25519 (ο αλγόριθμος αυτός
χρησιμοποιείται από το BigchainDB)
    const sender = new driver.Ed25519Keypair(
    fromHexadecimalStringtoUint8Array(userS.address_publicKey));

    //δημιουργώ το genesis block στην τοπική μου βάση που
αντιστοιχεί σε έγγραφο της συλλογής Blockchain
    const genesis = new Blockchain({
```

```
        message: 'This is the genesis block',
        senderId: '',
        recipientId: '',
        timestamp: GetDateTimeNow(),
        previousBlockHash: '',
        currentBlockHash:
'd41cbddc5fe1aeb84e967773845f21a63be45bb5a4e758e57bb453cd4b4768a811
cdaaf801ecc5cfb0d826bc20bf14f47d064156c2ce632dc8044ca64bbf4aba',
        index: 0,
        name: 'BlockchainMessenger_v1'
    })

    //δημιουργώ την συναλλαγή για το BigchainDB σύμφωνα με τις
προκαθορισμένες εντολές
    const transaction = driver.Transaction.makeCreateTransaction(
        genesis,
        null,
        [ driver.Transaction.makeOutput(
                driver.Transaction.makeEd25519Condition
                (sender.publicKey))
        ],
        sender.publicKey
    );

    //υπογράφεται η συναλλαγή με το ιδιωτικό κλειδί του αποστολέα
    const transactionSigned = driver.Transaction.signTransaction(
                              transaction, sender.privateKey);

    //στέλνεται η συναλλαγή στο blockchain του BigchainDB και
αποθηκεύεται
    conn.postTransactionCommit(transactionSigned);


    await genesis.save();
}
```

δ) αν υπάρχει το blockchain μας, τότε πρέπει να επικυρωθεί με την μέθοδο VerifyTheBlockchain(...) , η οποία ελέγχει διάφορες περιπτώσεις που το blockchain μας δεν είναι έγκυρο και τότε θέτει την μεταβλητή validity σε false. Πρακτικά, με τη σειρά που υπάρχουν στον κώδικα ελέγχεται 1) αν έχουν διαγραφεί τα πάντα από την τοπική βάση, 2) αν έχει διαγραφεί μόνο το genesis μπλοκ, 3) αν υπάρχουν τουλάχιστον 2 κουτιά στην αλυσίδα μας, τότε πρέπει να ελεγχθούν τα hashes όλων των κουτιών για το αν είναι σωστά και 4) ελέγχεται αν η τοπική αλυσίδα μας έχει ίδιο μήκος με την αλυσίδα από το BigchainDB (δηλαδή ελέγχεται αν έχει διαγραφεί κάποιο, οποιοδήποτε κουτί από την τοπική μας βάση). Αν δεν ισχύει τίποτα από αυτά τότε το blockchain μας είναι έγκυρο και επιστρέφεται true, αλλιώς false.

```
const VerifyTheBlockchain = async(blockchain) => {
    var validity = true;

    // περιμένω να πάρω την απάντηση από το promise searchAssets
για το αντικείμενο που περιέχει το blockchain στο BigchainDB
    const res = await conn.searchAssets('BlockchainMessenger_v1');

    // αν εχουν διαγραφει τα παντα και δεν υπάρχει τίποτα
    if(blockchain.length === 0) {
        validity = false;
    }
    //η αν έχει διαγραφεί το genesis block
    else if(blockchain.length > 0){
        if(blockchain[0].index !== 0){
            validity = false;
        }
    }

    // από το i=1, γιατί εδώ δεν μας ενδιαφέρει το genesis block
(το ελέγξαμε παραπάνω). Έτσι, ελέγχω μονο αν η αλυσίδα έχει πάνω
από 2 στοιχεια δηλαδή το length είναι >= 2. Αν έχει μονο το genesis
δεν χρειάζεται να ελέγχουμε ξανά
    if(blockchain.length>=2){
        for(var i=1; i<blockchain.length; i++){

            var ThisBlock = blockchain[i];
            var PreviousBlock = blockchain[i-1];

            // εδώ γίνεται ο έλεγχος αν είναι έγκυρο το blockchain,
με την διασταύρωση των hashes
            if( ThisBlock.previousBlockHash !==
                 PreviousBlock.currentBlockHash ||
                 ThisBlock.currentBlockHash !==
                 ComputeTheHashOfThisBlock(ThisBlock.message,
                 ThisBlock.previousBlockHash, ThisBlock.timestamp,
                 ThisBlock.senderId, ThisBlock.recipientId,
                 ThisBlock.index)
             ){
                 validity = false;
                 break;
            }
        }
    }

    // αυτό λοιπόν πρόκειται για ένα αντικείμενο promise, αν όλα
πήγαν καλά και περιέχει δεδομένα (δηλαδή το blockchain μου) και
μέχρι στιγμής το blockchain είναι έγκυρο, τότε πρέπει να ελέγχω αν
το μήκος από το response (τον πίνακα blockchain, που είναι όμως στο
bigchaindb και είναι αξιόπιστο) είναι ίσο με το μήκος του
blockchain μας στην τοπική βάση. Αν δεν είναι ίδιο, σημαίνει ότι
έχει διαγραφεί κάποιο block από την τοπική βάση και πρέπει να γίνει
ανακατασκευή, άρα η εγκυρότητα να πάει στο false
    if(res && validity){
        if(res.length !== blockchain.length){
            validity = false;
        }
    }
```

```
    console.log(validity);
    return validity;
}
```

Αν η αλυσίδα μας δεν είναι έγκυρη (η παραπάνω συνάρτηση επέστρεψε false), τότε πρέπει να ανακατασκευαστεί με τη συνάρτηση ReconstructTheBlockchain() εμφανίζοντας παράλληλα μήνυμα ενημέρωσης προς τον χρήστη μέσω της μεθόδου socket informTheUserWhenEntering.

```
const ReconstructTheBlockchain = async() => {

    //πρέπει να ξαναφέρω από το bigchaindb όλες τις συναλλαγές που συγκεντρωτικά δημιουργούν το τοπικό μου blockchain. Πρακτικά, κάθε συναλλαγή στο bigchaindb είναι και ένα block στο τοπικό blockchain μου. οπότε διαγράφω τα πάντα που εχουν απομείνει και θα το φέρω ξανά με την σωστή του μορφή
    const removed = await Blockchain.deleteMany({});

    // η τρέχουσα μέθοδος καλείται μόνο όταν έχει αποτύχει ο έλεγχος εγκυρότητας. Δηλαδή είτε δεν υπάρχει τίποτα στην τοπική βάση μας στον πίνακα blockchain (διαγράφηκαν όλα τα blocks και το removed.deletedCount πρακτικά είναι 0) είτε διαγράφηκαν όλα τα εναπομείναντα blocks
    if(removed.deletedCount>=0){
        conn.searchAssets('BlockchainMessenger_v1').then( response => {
            for(let i=0; i<response.length;i++){
                let blockdata = response[i].data;

                const block = new Blockchain({
                    message: blockdata.message,
                    senderId: blockdata.senderId,
                    recipientId: blockdata.recipientId,
                    timestamp: blockdata.timestamp,
                    previousBlockHash: blockdata.previousBlockHash,
                    currentBlockHash: blockdata.currentBlockHash,
                    index: blockdata.index,
                    name: blockdata.name
                })
                block.save();
            }
        })
    }
}
```

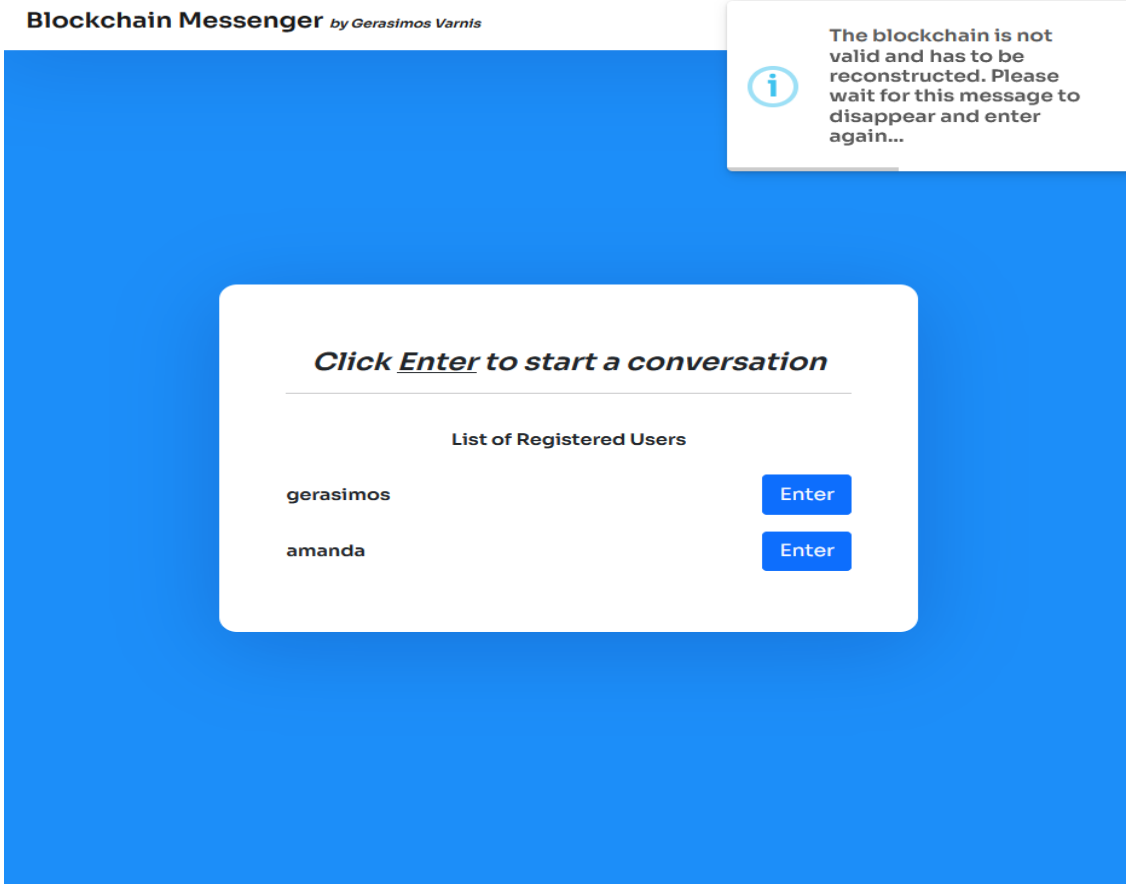
Σε αυτή την περίπτωση έχει γίνει η ανακατασκευή και καλείται η συνάρτηση informTheUserWhenEntering του socket, η οποία αντιστοιχεί στον κώδικα του αρχείου conversation.js

```
//αν όταν κάνουμε enter σε μία συζήτηση, εξακριβωθεί από τους ελέγχους στο backend ότι το blockchain στην βάση μας δεν είναι έγκυρο, τότε γίνεται ανακατεύθυνση στο lobby όπου εμφανίζεται
```

```
μήνυμα ότι ανακατασκευάζεται το blockchain και πρέπει να γίνει
enter ξανά.
       socket.on('informTheUserWhenEntering', () => {
          window.location.href = 'http://localhost:3000/lobby';
          localStorage.setItem('reconstructedWhenEntering', true);
       })
```

από όπου μεταφερόμαστε στο αρχείο lobby.js και συγκεκριμένα στο ακόλουθο σημείο, όπου ενημερώνεται ο χρήστης ότι ανακατασκευάστηκε η αλυσίδα μας και θέτουμε το item 'reconstructedWhenEntering' σε false για την επόμενη αντίστοιχη περίπτωση.

```
//μήνυμα ενημέρωσης του χρήστη για την περίπτωση που το τοπικό
blockchain μας δεν είναι έγκυρο όταν ο χρήστης κάνει enter στη
συζήτηση. αυτό εδώ είναι το τελευταίο βήμα. ξεκινάμε από το
backend, πάμε στο frontend στο αρχείο conversation.js και από εκεί
καταλήγουμε εδώ.
  if(localStorage.getItem('reconstructedWhenEntering') === 'true'){
      showMessage('info','The blockchain is not valid and has to be
                  reconstructed. Please wait for this message to
                  disappear and enter again...');
      localStorage.setItem('reconstructedWhenEntering', false);
    }
```



Εικόνα 20 - Ανακατασκευή blockchain κατά την είσοδο σε συνομιλία

Έπειτα, στο σημείο του server.js μετά την κλήση informTheUserWhenEntering πρέπει να βρούμε τον αποστολέα και τον παραλήπτη από την βάση μας, να δημιουργήσουμε τα ζευγάρια των κλειδιών για τον καθένα (με βάση τα οποία θα γίνει η αποκρυπτογράφηση των συναλλαγών/μηνυμάτων) και να οριστεί κάποιο onetimecode (που θα είναι πάντα το ίδιο και θα είναι 8 χαρακτήρες) για την αποκρυπτογράφηση. Αμέσως μετά βρίσκω τα blocks που περιέχουν τα μηνύματα μεταξύ των 2 χρηστών είτε αυτά που έχει στείλει ο ένας και τα παρέλαβε ο άλλος είτε το αντίστροφο και εντέλει βρίσκω τα κουτιά που περιέχουν όλα τα μηνύματα μεταξύ των 2 χρηστών, ενώ ορίζω έναν πίνακα userMessages για να τα αποθηκεύσω και να τα επιστρεψω μετά στο front-end. Αν υπάρχει έστω και ένα μήνυμα, τότε αποκρυπτογραφούνται ανοίγει το nacl.box χρησιμοποιώντας το ανάλογο δημόσιο κλειδί του αποστολέα και το onetimecode και το ιδιωτικό κλειδί του παραλήπτη. Μετατρέπεται το μήνυμα σε αναγνώσιμο format, δημιουργείται το αντικείμενο του μηνύματος και προστίθεται/ γίνεται push στον πίνακα μηνυμάτων.

```
        const userS = await User.findOne({_id: socket.userId}); //
αποστολέας
        const userR = await User.findOne({_id: userId}); //
παραλήπτης

        const senderKeyPair =
nacl.box.keyPair.fromSecretKey(fromHexadecimalStringtoUint8Array(us
erS.address_publicKey));
        const recipientKeyPair =
nacl.box.keyPair.fromSecretKey(fromHexadecimalStringtoUint8Array(us
erR.address_publicKey));

        const oneTimeCode1 = recipientKeyPair.publicKey.slice(8);
        const oneTimeCode2 = senderKeyPair.publicKey.slice(8);

// βρίσκω τα blocks που περιέχουν τα μηνύματα μεταξύ των 2 χρηστών
είτε αυτά που χει στείλει ο ένας και τα παρέλαβε ο άλλος είτε το
αντίστροφο
   const blocks = await Blockchain.find(
    {$or:[
          {senderId: socket.userId, recipientId: userId},
          {senderId: userId, recipientId: socket.userId}
    ] })
    const userMessages = [];
    if(blocks.length>=1){
       for(var block of blocks){
```

```
        let decodedMessage = block.senderId ===  socket.userId ?
    nacl.box.open(fromHexadecimalStringtoUint8Array(block.message),
    oneTimeCode1, senderKeyPair.publicKey,
    recipientKeyPair.secretKey) :
    nacl.box.open(fromHexadecimalStringtoUint8Array(block.message),
    oneTimeCode2, recipientKeyPair.publicKey,
    senderKeyPair.secretKey);
        let readableMessage=nacl.util.encodeUTF8(decodedMessage);

        const message = {
            message: readableMessage,
            username: block.senderId === socket.userId ?
            userS.name : userR.name,
            userSender: block.senderId
        }
        userMessages.push(message);
      }

      if(userMessages && userS && userR){
        SocketIO.to(conversationId).emit(
         'getAllMessages',{allMessages: userMessages});
       }
    }
```

Αν λοιπόν περιέχονται μηνύματα στον πίνακα και οι χρήστες (αποστολέας, παραλήπτης) έχουν βρεθεί τότε καλείται η μέθοδος getAllMessages, με το συγκεκριμένο conversationId και ως παράμετρο allMessages τον πίνακα των μηνυμάτων. Η διαδικασία αυτή τελειώνει με επιστροφή στο front-end και στην μέθοδο getAllMessages του αρχείου conversation.js, όπου ενημερώνεται η κατάσταση των μηνυμάτων μέσω της setMessages() και έτσι πλέον τα μηνύματα που επιστρέψαμε από το back-end, υπάρχουν στο τρέχον state του messages[].

```
//φέρνει το ιστορικό όλων των μηνυμάτων από το blockchain της βάσης
            socket.on('getAllMessages', ({allMessages}) => {
                var newMessages = [];
                if(allMessages){
                    for (const msg of allMessages) {
                        const message = {
                            name: msg.username,
                            userId: msg.userSender,
                            message: msg.message
                        }
                        newMessages.push(message);
                    }
                    setMessages(newMessages);
                }
            });
```

Τελικά, υπάρχει μία return πριν κλείσει η τρέχουσα useEffect και είναι μία ειδική περίπτωση για το όταν θέλω να κάνω κάτι όταν γίνει

unmount το component, δηλαδή όταν φεύγουμε από το συγκεκριμένο view. Εμείς θέλουμε να φεύγει από την συνομιλία ο χρήστης που κλείνει τον φυλλομετρητή ή πατάει το back, πάει στο lobby, κάνει logout ή ανανέωση.

```
return () => {
            if(socket){
                socket.emit('leaveConversation', {
                    conversationId: conversationID,
                })
            }
        }
```

Οδηγούμαστε και πάλι στο αρχείο server.js και συγκεκριμένα στον παρακάτω κώδικα, όπου ο χρήστης βγαίνει από τη σύνδεση socket με το αντίστοιχο conversationId. Σε αυτό το σημείο τελειώνει η πρώτη useEffect.

```
socket.on('leaveConversation', (conversationId) => {
        socket.leave(conversationId);
        console.log('left the conversation ' +
conversationId.conversationId) // όταν γίνεται emit το
leaveConversation στο frontend επιστρέφει conversationId σαν
αντικείμενο με ιδιότητα το conversationId και όχι απλά σαν string
    })
```

- Η δεύτερη useEffect του αρχείου conversation.js φαίνεται παρακάτω και καλείται κάθε φορά που γίνεται μία αλλαγή στην κατάσταση των messages, δηλαδή κάθε φορά που στέλνεται μήνυμα μεταξύ των 2 χρηστών.

```
    //κάθε φορά που αλλάζει το state των μηνυμάτων μπαίνει στην
useEffect. ΟΜΩΣ, προσοχή. εδώ δεν καλείται η μέθοδος, αλλά
αρχικοποιείται/δηλώνεται η newConvMessage. Πρακτικά, αυτή θα
καλείται όταν γίνεται το emit/η κλήση από το backend, μετά την
κλήση της παραπάνω μεθόδου conversationMessage. Αυτό γίνεται, διότι
πριν σταλεί κάποιο μήνυμα υπάρχει διαδικασία στο αρχείο server.js
(ελέγχεται το blockchain μας, κρυπτογραφείται το μήνυμα, γίνεται η
συναλλαγή bigchaindb και αποθηκεύεται στο blockchain του bigchaindb
επιτυχώς και αν όλα αυτά πάνε καλά, τότε ετοιμάζεται η
εγγραφή/αποθήκευση του μηνύματος/block στην τοπική μας βάση στον
πίνακα blockchain, αποκρυπτογραφείται το μήνυμα και γίνεται η κλήση
της μεθόδου newConvMessage (και ερχόμαστε εδώ κάτω) με την
παράλληλη ασύγχρονη αποθήκευση του block στη βάση)
React.useEffect(() => {

        if(socket){
            socket.on('newConvMessage', (message) => {
                const newMessages = [...messages, message];
                setMessages(newMessages);
            });
```

```
            // αν κατά την διάρκεια της συνομιλίας και συγκεκριμένα
όταν πατήσω την αποστολή, εξακριβωθεί από τους ελέγχους στο backend
ότι το blockchain στην βάση μας δεν είναι έγκυρο, τότε βγαίνει
μήνυμα ενημέρωσης του χρήστη 'να περιμένει όσο το μήνυμα αυτό
διαρκεί για να γίνει ανακατασκευή του blockchain' περίπου 5
δευτερόλεπτα.
            socket.on('informTheUserWhenSending', () => {
                showMessage('info','The blockchain is not valid and
has to be reconstructed. Please wait for this message to disappear
and try again...');
            })
        }
},[messages])

return (
    <div>
     {/* εδώ υπάρχει ο έλεγχος για το userToken, αν είναι null.
δηλαδή πρέπει να έχει γίνει πετυχημένη είσοδος του χρήστη για να
μπορέσει να δεί την σελίδα αυτή. αν πχ κάποιος πατήσει τη σελίδα
χωρίς να έχει μπεί στην εφαρμογή ή έχει κάνει logout, τότε θα
εμφανίζεται μήνυμα ότι δεν έχεις την εξουσιοδότηση για να δεις την
σελίδα αυτή. αν υπάρχει το token, τότε εμφανίζεται κανονικά. */}
       {localStorage.getItem('userToken') !== 'null' &&
        localStorage.getItem('userToken') !== null ?
           <div>
             <div className="conversationBox">
               <div className="receiver">
                   <span>{receiverName}</span>
               </div>
               <br />
               <div className="conversationInterior">
                   {messages.map((msg, index) => (
                     <div key={index} className="message">
                       <span className={receiverName !== msg.name ?
                        "me" : "others"}>{receiverName !== msg.name
                        ? 'Me' : msg.name}: </span> {msg.message}
                                 </div>
                          ))}
                    </div>
                    <div className="conversationHandlers">
                       <div>
                         <input type="text" name='message'
                            placeholder='Type here...'
                            ref={msgReference}
                            onKeyPress={handleEnter} />
                       </div>
                       <div>
                         <button className="btn btn-primary btn-
                            block"
                            onClick={sendMessage}>Send
                         </button>
                       </div>
                     </div>
                </div>
             </div>
              :
             <div>
```

```
                    <br />
                    <h3 style={{textAlign: 'center'}}>You are not
                        authorized to view this page.<br /><br/>
                        Please login or sign-up.</h3>
                </div>
            }
        </div>
    );
```

Κάθε φορά που ο χρήστης γράφει ένα μήνυμα και πατάει το κουμπί *Send* ή το Enter στο πληκτρολόγιό του καλείται η μέθοδος sendMessage() (του αρχείου conversation.js) για να σταλεί το μήνυμα.

```
// στέλνει το μήνυμα και όταν πατάω το enter
    const handleEnter = e => {
        if (e.charCode === 13) {
            sendMessage();
          }
    }

    //η μέθοδος που καλείται όταν πατάμε send ή enter για να
στείλουμε το μήνυμα μας. ελέγχουμε αν υπάρχει το socket αρχικά (για
να γίνει η σύνδεση με το backend) και κάνω emit (καλώ) την μέθοδο
conversationMessage που έχει οριστεί στο backend/ αρχείο server.js.
Περνάω σαν παραμέτρους το id της συνομιλίας, το μήνυμα kai το id
του χρήστη που λαμβάνει το μήνυμα. Εκεί είναι που εν τέλει αν όλα
πάνε καλά καλείται η μέθοδος newConvMessage
    const sendMessage = () => {
        if(socket){
            socket.emit('conversationMessage', {
                conversationId: conversationID,
                message: msgReference.current.value,
                userId: userID
            });

            //μετά την αποστολή του μηνύματος αρχικοποιώ την
αναφορά μου για το τι γράφει ο χρήστης, ώστε να είναι έτοιμο για
την επόμενη αποστολή
            msgReference.current.value = "";
        }
    }
```

Τα σχόλια εξηγούν επακριβώς πως λειτουργεί η συνάρτηση και καλείται η conversationMessage, που βρίσκεται στο back-end στο αρχείο server.js όπου περνάμε σαν παραμέτρους το id της συνομιλίας *conversationID*, το μήνυμα που στέλνει ο χρήστης μέσω της αναφοράς από το αντίστοιχο input *msgReference.current.value* και το id του παραλήπτη χρήστη *userID*. Επομένως, η διαδικασία αποστολής μηνύματος έχει ως εξής:

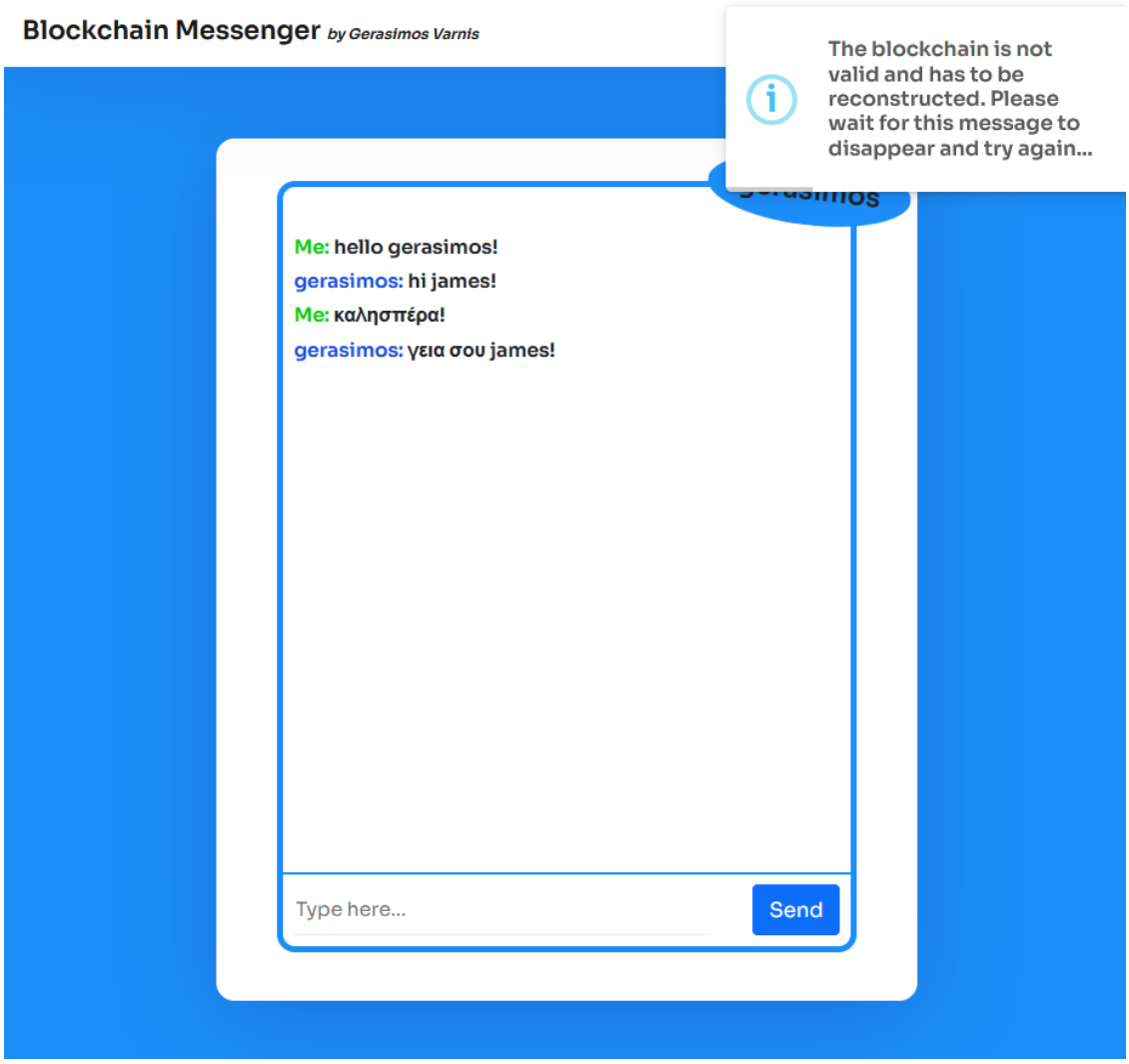
```
socket.on('conversationMessage', async ({conversationId, message,
userId}) => {
```

```
        const userS = await User.findOne({_id: socket.userId}); //
αποστολέας

        const userR = await User.findOne({_id: userId}); //
παραλήπτης

        var blockchain = await Blockchain.find(); // επιστρέφει όλο
blockchain


        //μονο αν το μήνυμα δεν είναι κενό τότε κάνω emit μια νέα
        μέθοδο newMessage και εδώ θα ελέγχω αν είναι έγκυρο το
        blockchain
        if(message){
            var valid = await VerifyTheBlockchain(blockchain);
            var validAfterRecons;

            console.log('is chain valid?' + valid);
            if(!valid){
             const reconstructed= await ReconstructTheBlockchain();
             if(reconstructed){
               validAfterRecons= VerifyTheBlockchain(blockchain);
             }

             SocketIO.to(conversationId).emit(
                 'informTheUserWhenSending', {});

             blockchain = await Blockchain.find();
            }

            if(valid || validAfterRecons){
             console.log('chain is valid');
                // keypair αποστολέα για την συναλλαγή/transaction
                 στο bigchaindb
             const sender = new driver.Ed25519Keypair(
       fromHexadecimalStringtoUint8Array(userS.address_publicKey));

                // τα keypairs που θα χρησιμοποιηθούν για την
κρυπτογράφηση σύμφωνα με το δημόσιο κλειδί που έχει δοθεί στον κάθε
χρηστη. αυτά δεν αποθηκεύονται για λόγους ασφαλείας, αλλα κάθε φορα
γίνονται generate με βάση το δημόσιο κλειδί που υπάρχει στη βάση
             const senderKeyPair = nacl.box.keyPair.fromSecretKey(
       fromHexadecimalStringtoUint8Array(userS.address_publicKey));

             const recipientKeyPair=nacl.box.keyPair.fromSecretKey(
       fromHexadecimalStringtoUint8Array(userR.address_publicKey));
            const oneTimeCode= recipientKeyPair.publicKey.slice(8);

            //ποιο είναι το μήνυμα που γράφει ο χρήστης
            const unencryptedMessage = message;

                //κρυπτογράφησε το
            const encryptedMessage = nacl.box(
                 nacl.util.decodeUTF8(unencryptedMessage),
                 oneTimeCode,
                 recipientKeyPair.publicKey,
                 senderKeyPair.secretKey
```

```
            );

            const hexEncryptedMessage =
              fromUint8ArraytoHexadecimalString(encryptedMessage);

            const Timestamp = GetDateTimeNow();
            const PreviousBlockHash = blockchain[blockchain.length
-1].currentBlockHash;
            const BlockIndex = blockchain[blockchain.length -
1].index + 1;
            const CurrentBlockHash =
ComputeTheHashOfThisBlock(hexEncryptedMessage, PreviousBlockHash,
Timestamp, userS._id, userR._id, BlockIndex);


            const assetData_BlockData = {
                message: hexEncryptedMessage,
                senderId: userS._id ,
                recipientId: userR._id,
                timestamp: Timestamp,
                previousBlockHash: PreviousBlockHash,
                currentBlockHash: CurrentBlockHash,
                index: BlockIndex,
                name: 'BlockchainMessenger_v1'
            }

            const transaction =
              driver.Transaction.makeCreateTransaction(
                    assetData_BlockData,
                    null,
                    [ driver.Transaction.makeOutput(
                        driver.Transaction.makeEd25519Condition(
                        sender.publicKey))
                    ],
                    sender.publicKey
                );

             const transactionSigned=
driver.Transaction.signTransaction(transaction, sender.privateKey);

             conn.postTransactionCommit(transactionSigned);

             const newBlock = new Blockchain(assetData_BlockData);



             let decodedMessage = nacl.box.open(
            fromHexadecimalStringtoUint8Array(hexEncryptedMessage),
oneTimeCode, senderKeyPair.publicKey, recipientKeyPair.secretKey);

             let readableMessage =
                 nacl.util.encodeUTF8(decodedMessage);

               // είναι από τη μεριά αυτού που στέλνει, άρα userId
και name αυτού που το στέλνει
                SocketIO.to(conversationId).emit('newConvMessage',
                {
```

```
                userId: userS._id,
                name: userS.name,
                message: readableMessage
            });

            await newBlock.save();
        }
    }
})
```

1. Βρίσκουμε τον αποστολέα, τον παραλήπτη και την αλυσίδα μας από την τοπική βάση και τα αποθηκεύουμε σε 3 μεταβλητές *userS, userS, blockchain* αντίστοιχα.
2. Αν το μήνυμα που έχουμε περάσει από το front-end δεν είναι κενό τότε α) ελέγχεται η εγκυρότητα του blockchain με την μέθοδο VerifyTheBlockchain(blockchain) που περιγράφηκε ήδη.
3. Αν δεν είναι έγκυρη, τότε ανακατασκευάζεται η αλυσίδα μας ξανά με την μέθοδο ReconstructTheBlockchain() που προαναφέρθηκε και όταν γίνει αυτό ξανά ελέγχεται η εγκυρότητα, ενώ παράλληλα καλείται και η informTheUserWhenSending (που βρίσκεται μέσα στην 2η useEffect() του conversation.js) για να ενημερωθεί ο χρήστης ότι το blockchain ανακατασκευάζεται (Εικόνα 21).
4. Αν η αλυσίδα είναι έγκυρη εξαρχής ή είναι έγκυρη μετά την ανακατασκευή, τότε δημιουργείται το keypair του αποστολέα για την συναλλαγή/transaction στο bigchaindb σύμφωνα με το documentation και επίσης δημιουργούνται 2 keypairs για τον αποστολέα, παραλήπτη σύμφωνα με τον αλγόριθμο του nacl ώστε να γίνει η κρυπτογράφηση του μηνύματος, όπως επίσης και το onetimecode για τον παραλήπτη.
5. Κρυπτογραφείται το μήνυμα και προκύπτει το encryptedMessage (αυτό ειναι σε Uint8Array μορφή) και έπειτα μετατρέπεται σε δεκαεξαδική μορφή hexEncryptedMessage, για να είναι πιο εύκολα διαχειρίσιμο.



Εικόνα 21 - Ανακατασκευή blockchain κατά την αποστολή μηνύματος

6. Δημιουργούνται τα δεδομένα της συναλλαγής/μηνύματος για το BigchainDB που θα χαρακτηρίζουν και το αντίστοιχο κουτί στην τοπική μας βάση Timestamp, previousBlockHash, BlockIndex, CurrentBlockHash. Εδώ χρησιμοποιούνται και άλλες 2 μέθοδοι του αρχείου server.js, οι GetDateTimeNow() και ComputeTheHashOfThisBlock(...) που επιστρέφουν την τωρινή ημερομηνία-ώρα και δημιουργούν το hash του κουτιού/συναλλαγής.

```
const GetDateTimeNow = () => {
    var DateTime = new Date();
    var date = DateTime.getFullYear()+ '/' +
(DateTime.getMonth() + 1) + '/' + DateTime.getDate();
```

```
    var time = DateTime.getHours() + ":" +
DateTime.getMinutes() + ":" + DateTime.getSeconds();
    return date + ' ' + time;
}

const ComputeTheHashOfThisBlock = (MessageTransactions,
PreviousBlockHash, Timestamp, SenderId, RecipientId, Index)
=> {
    return sha512(JSON.stringify(MessageTransactions) +
PreviousBlockHash + Timestamp + SenderId + RecipientId +
Index).toString();
}
```

7. Δημιουργείται το αντικείμενο assetData_BlockData που θα είναι η συναλλαγή που θα αποθηκευτεί στο BigchainDB και το block που θα αποθηκευτεί στην τοπική μας βάση. Αυτό το αντικείμενο αντιστοιχεί σε ένα μήνυμα.
8. Δημιουργείται η συναλλαγή/transaction για το BigchainDB σύμφωνα με το documentation, υπογράφεται με το ιδιωτικό κλειδί του αποστολέα και στέλνεται στο BigchainDB για να καταχωρηθεί.
9. Σχηματίζεται το νέο block για την τοπική μας βάση, που θα περιέχει το αντικείμενο assetData_BlockData και πρόκειται για την μεταβλητή newBlock.
10. Αποκρυπτογραφείται το μήνυμα και έπειτα μετατρέπεται στην σωστή κωδικοποίηση UTF8.
11. Καλείται η newConvMessage του αρχείου conversation.js στο front-end, περνώντας το id του αποστολέα, το όνομα του και το μήνυμα (πλέον σε αναγνώσιμη μορφή). Πρόκειται για το ίδιο κανάλι socket με το οποίο έγινε η αρχική conversationMessage, καθώς έχουμε το ίδιο conversationId.
12. Αποθηκεύεται το νέο κουτί/νέο μήνυμα newBlock στην τοπική μας βάση.

Πλέον είμαστε ξανά στο front-end και στην 2η useEffect, αφού έχει κληθεί η newConvMessage και εκεί ενημερώνω την κατάσταση των μηνυμάτων για το νέο μήνυμα με την setMessages(). Εφόσον άλλαξε η κατάσταση των μηνυμάτων, καλείται η useEffect και τότε γίνεται render / εκ νέου εμφάνιση του component, του view το οποίο περιέχει

τελικά το νέο μήνυμα που έστειλε ο χρήστης. Έτσι, έχει σταλεί το μήνυμα και έχει εμφανιστεί και στους 2 χρήστες που συμμετέχουν στη συζήτηση.

- Τα μηνύματα εμφανίζονται μόνο αν έχει ο χρήστης έγκυρο userToken και η εμφάνιση πετυχαίνεται με ένα map στην σταθερά των μηνυμάτων/messages. Για κάθε μήνυμα μέσα στον πίνακα messages εμφανίζεται ένα span element το οποίο έχει πράσινο η μπλέ χρώμα (css κλάση me/others), ανάλογα με το αν το όνομα στο κάθε μήνυμα αντιστοιχεί στο receiverName (που είχαμε πάρει από τις παραμέτρους του match.params.conversationParameters στην αρχή) ή όχι, το όνομα του αποστολέα η του παραλήπτη αριστερά από το μήνυμα με την ίδια συνθήκη και το περιεχόμενο του μηνύματος, το κείμενο που θα υπάρχει στο msg.message.

```
<div className="conversationInterior">
  {messages.map((msg, index) => (
    <div key={index} className="message">
      <span className={receiverName !== msg.name ? "me":"others"}>
          {receiverName !== msg.name ? 'Me' : msg.name}:
      </span>
      {msg.message}
    </div>
   ))}
</div>
```

4.4 Διαχείριση σφαλμάτων και αποσύνδεση χρήστη

Η αποσύνδεση του χρήστη από την εφαρμογή γίνεται πατώντας το κουμπί Logout πάνω δεξιά στα views lobby και conversation, ενώ αντιστοιχεί στη μέθοδο disconnect του αρχείου App.js.

```
// εδώ γίνεται η χειροκίνητη αποσύνδεση του χρήστη από την εφαρμογή, ήτοι το
logout. Αν λοιπόν ο χρήστης πατήσει το κουμπί logout παίρνω τα στοιχεία της
σύνδεσης socket από το backend, σύμφωνα με το αναγνωριστικό token του χρήστη
και αν έχει τιμή το state του socketIO (δηλαδή δεν είναι null), τότε κάνω
την αποσύνδεση με το αντίστοιχο μήνυμα
  const disconnect = () => {
    const socket = IO("http://localhost:8000", {
        query: {
            token: localStorage.getItem('userToken'),
        },
      });
      if(socketIO){
        socket.disconnect();
        localStorage.setItem('userToken', null);
```

```
        setSocketIO(null);
        showMessage('success', 'You have logged out.');
      }
  }
```

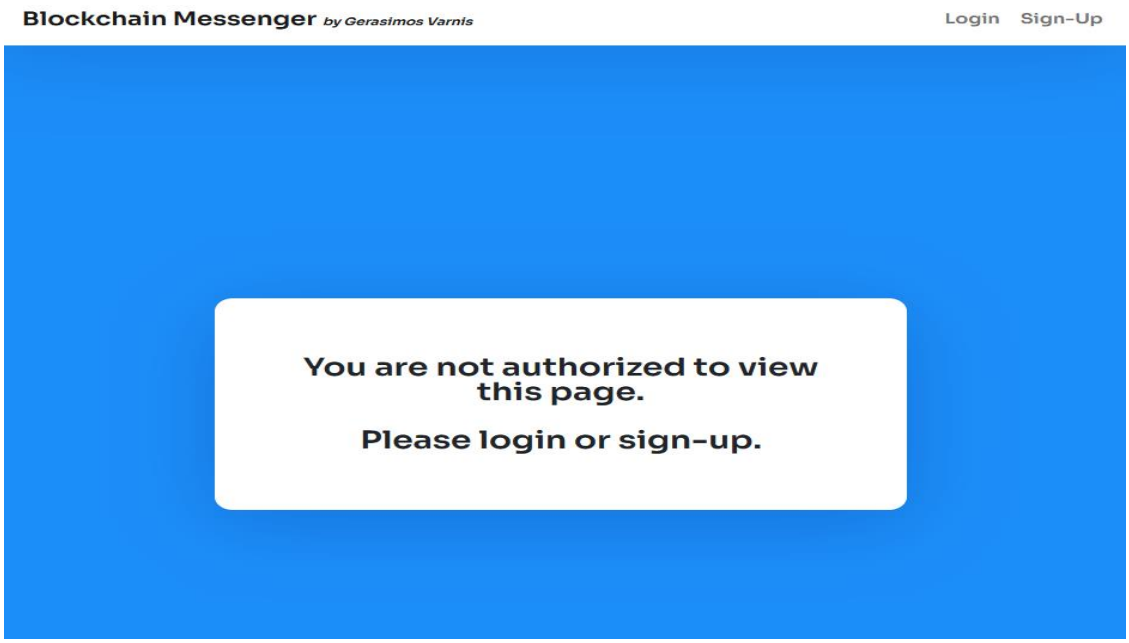
Αν γίνει η αποσύνδεση ή κάποιος ανοίξει την εφαρμογή χωρίς να πραγματοποιήσει είσοδο, τότε δεν έχει λάβει το userToken με το οποίο αναγνωρίζεται στην εφαρμογή και πιστοποιείται. Σε αυτές τις περιπτώσεις αν κάποιος μεταβεί στα url http://localhost:3000/lobby ή για παράδειγμα http://localhost:3000/conversation/6258797276236498_61965f50679cab98f4e7cf39_gerasimos, χειροκίνητα/πληκτρολογώντας ή πατώντας το κουμπί πίσω του φυλλομετρητή, δεν θα μπορεί να δεί την ιστοσελίδα και θα βγαίνει το αντίστοιχο μήνυμα. Έτσι, στα αρχεία conversation.js και lobby.js υπάρχει ο κώδικας,

```
:
              <div>
                <br />
                <h3 style={{textAlign: 'center'}}>You are not authorized to
view this page.<br /><br />Please login or sign-up.</h3>
              </div>
```

που αντιστοιχεί στην περίπτωση που δεν υπάρχει το userToken σύμφωνα με τη συνθήκη (το localStorage συμπεριφέρεται διαφορετικά ανάλογα τον browser, οπότε μπορεί μην έχει καθόλου τιμή ή να περιέχει την τιμή 'null' ως string).

```
localStorage.getItem('userToken') !== 'null' &&
localStorage.getItem('userToken') !== null ?
```

Το div που θα εμφανίζεται, φαίνεται στην παρακάτω εικόνα

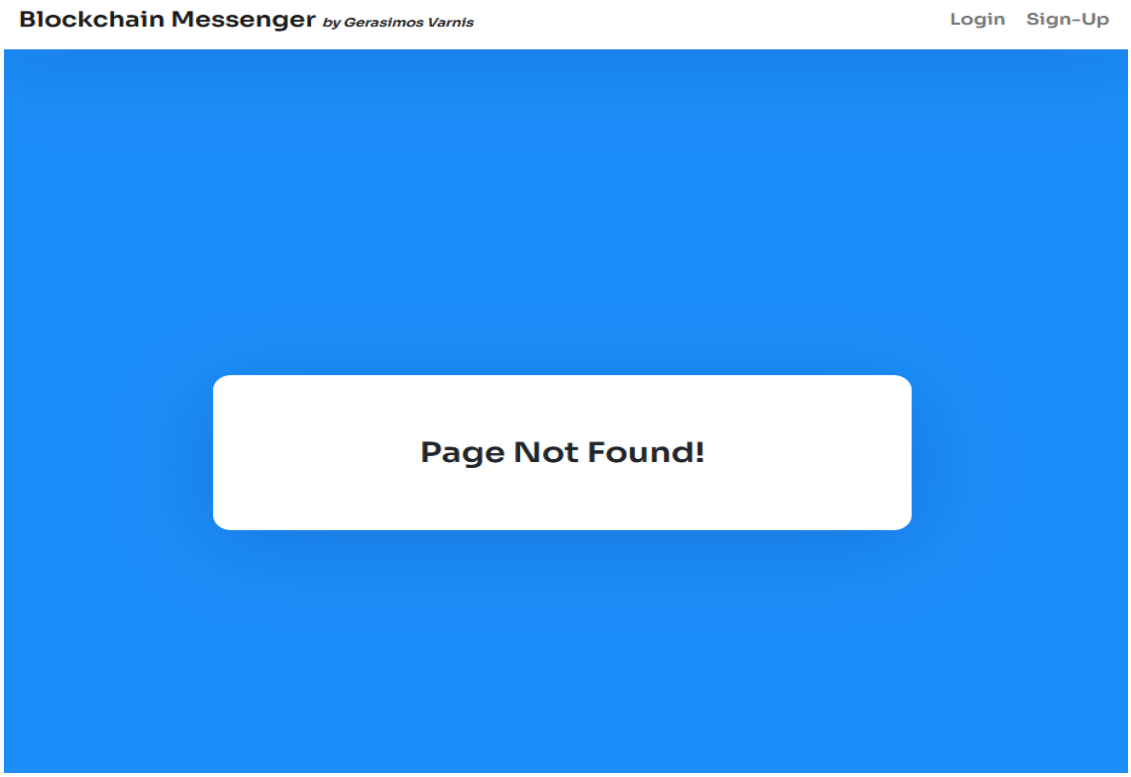


Εικόνα 22 - Μη εξουσιοδοτημένη πλοήγηση στα views lobby & conversation

Η τελευταία περίπτωση αφορά την λανθασμένη πληκτρολόγηση url από τον χρήστη και συγκεκριμένα στον κώδικα που ακολουθεί βλέπουμε πως αν ο χρήστης εισάγει κάτι διαφορετικό από τις λέξεις που υπάρχουν μέσα στο path, θα εμφανίζεται μήνυμα λάθους. Πρόκειται για regex, που σημαίνει αν ο χρήστης εισάγει κάτι διαφορετικό από τη λέξη login ή sign-up ή lobby ή conversation/:conversationParameters, που ακολουθείται από κενό ή όχι, κάποια τελεία ή όχι ή κάποια ακολουθία αλφαριθμητικών, τότε θα βγαίνει το μήνυμα μέσα στο h3 element. Πρακτικά, αν ο χρήστης πληκτρολογήσει το οτιδήποτε εκτός των login ή sign-up ή lobby ή conversation/:conversationParameters (όπου conversationParameters αντιστοιχεί στις παραμέτρους της συνομιλίας κάθε φορά) π.χ. http://localhost:3000/loby ή http://localhost:3000/loginn ή http://localhost:3000/sign-up and something, θα ενημερώνεται με μήνυμα. Θα μπορούσαμε να προσθέσουμε και άλλες περιπτώσεις, ωστόσο εξετάζεται η περίπτωση κατά λάθος πληκτρολόγησης και όχι τόσο η εσκεμμένα λάθος πληκτρολόγηση.

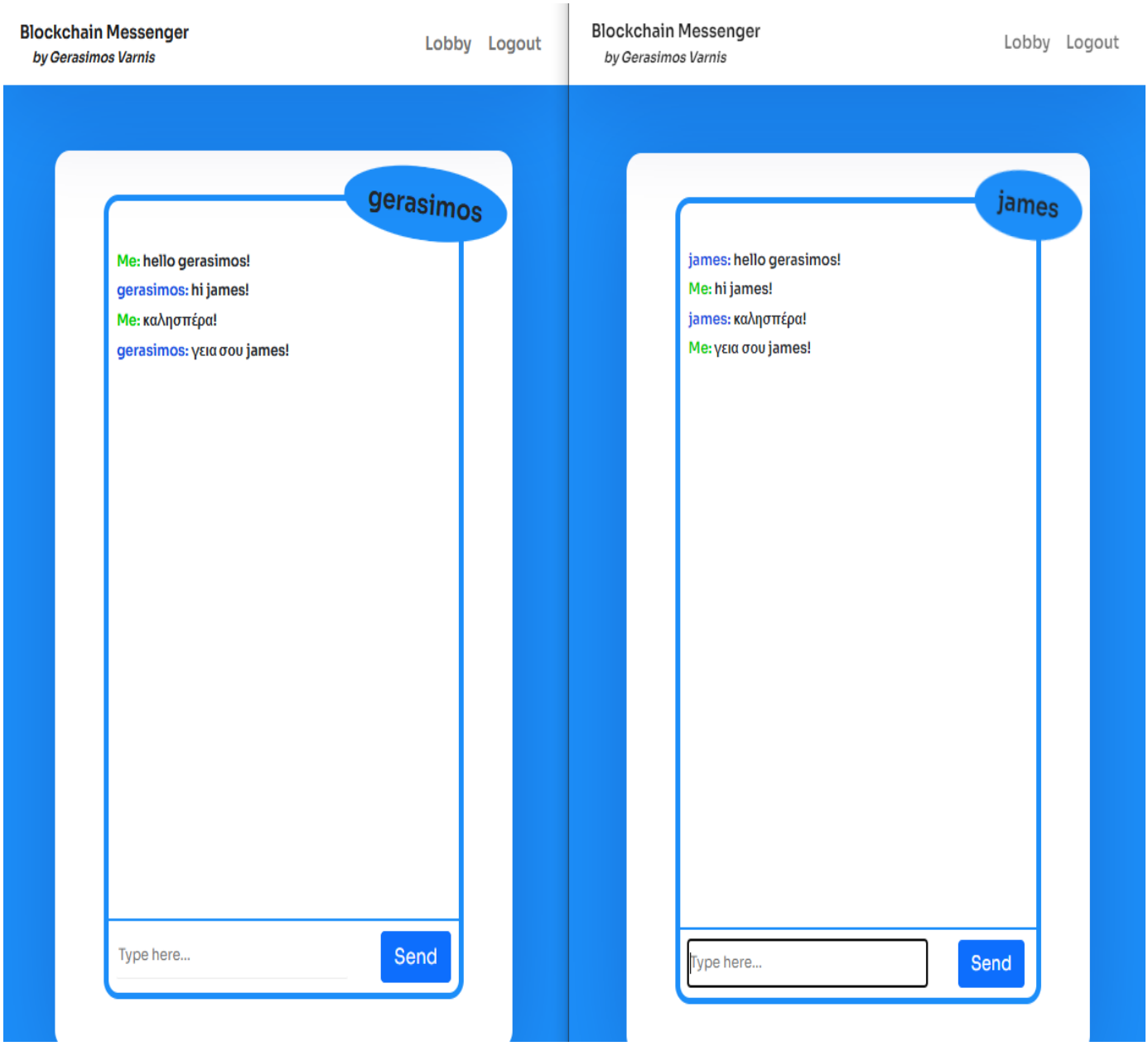
```
{/* εδώ γίνεται ο έλεγχος για το αν το url που εισάγει ο χρήστης δεν είναι
κάποιο από τα προηγούμενα αποδεκτά urls, τα οποία αντιστοιχούν και
περιγράφουν την λειτουργία της εφαρμογής μας. Αν εισάγει κάτι διαφορετικό
από αυτό που υπάρχει μέσα στην πρώτη παρένθεση του regex, τότε θα γίνεται
render ενός μόνο div με μήνυμα λάθους, πως η σελίδα δεν βρέθηκε  */}
            <Route exact
path={/^\/(?:login|signup|lobby|conversation\/:conversationParameters|)\s*.*
[A-Za-z0-9]+$/} render={() =>
              <div>
                <br />
                <h3 style={{textAlign: 'center'}}>Page Not Found!</h3>
              </div>
            }/>
```

Ο κώδικας μεταφράζεται στην εικόνα που ακολουθεί.



Εικόνα 23 - Λανθασμένη πληκτρολόγηση url

Σε αυτό το σημείο έχει ολοκληρωθεί το κεφάλαιο 4 και η επεξήγηση λειτουργίας της εφαρμογής ***Blockchain Messenger***. Ακολουθεί μία τελευταία εικόνα που δείχνει πως φαίνεται η συνομιλία μεταξύ 2 χρηστών, δηλαδή ταυτόχρονα σε 2 διαφορετικούς browsers.



Εικόνα 24 - Συνομιλία μεταξύ 2 χρηστών

Συμπεράσματα

Έχοντας φτάσει στην λήξη της παρούσας εργασίας, αξίζει να σημειωθεί πως η όλη διαδικασία αφ'ενός δείχνει τις τεράστιες προοπτικές της τεχνολογίας blockchain και τα πλεονεκτήματα που προσφέρονται σχετικά με την αξιοπιστία των μηνυμάτων που στέλνονται, ενώ αφ'ετέρου αποτελεί τη βάση για διερεύνηση πολυπλοκότερων σχεδιασμών καθώς σε επίπεδο enterprise μίας μεγάλης εταιρείας/οργανισμού θα μπορούσε να δημιουργηθεί ένα καινούριο, ιδιωτικό δίκτυο για τη λειτουργία του blockchain και να αντικατασταθεί με το BigchainDB. Γενικά, κάτι τέτοιο είναι εφικτό να υλοποιηθεί είτε για την ενδοεταιρική επικοινωνία είτε για την αποθήκευση πελατειακών αρχείων.

Η εφαρμογή θα μπορούσε να τροποποιηθεί ώστε να ανταλλάσσονται αρχεία μεταξύ χρηστών και για παράδειγμα να αποτελέσει ένα σημαντικό εργαλείο για την αξιόπιστη μεταφορά δεδομένων ανάμεσα στους εργαζομένους μίας επιχείρησης. Η τροποποίηση αυτή μπορεί εύκολα να γίνει με την ενσωμάτωση διαφόρων APIs στον κώδικα της εφαρμογής είτε με την μετατροπή του κώδικα ώστε στη θέση των μηνυμάτων να στέλνονται αρχεία μεταξύ των χρηστών. Επιπλέον, η λειτουργία της επικοινωνίας και ανταλλαγής μηνυμάτων ανά ζευγάρια χρηστών είναι κάτι που ανάλογα με τον επιθυμητό τρόπο χρήσης μέσα σε μία επιχείρηση, αλλάζει και επεκτείνεται σε γκρουπς πάνω από 2 άτομα ώστε η εφαρμογή να εξυπηρετεί σκοπούς ομαδικής συνομιλίας και έναν ψηφιακό χώρο που μπορούν να υπάρχουν καταχωρημένα αρχεία για την άμεση πρόσβαση από όλους τους χρήστες της εκάστοτε ομάδας ατόμων.

Ως γνωστόν ένα πολύ μεγάλο ποσοστό των επιχειρήσεων στον τομέα της πληροφορικής (και όχι μόνο) χρησιμοποιεί σε καθημερινή βάση εφαρμογές ανταλλαγής μηνυμάτων για την επικοινωνία των εργαζομένων της και ακόμα περισσότερο τώρα στην εποχή της πανδημίας που καθιερώθηκε η απομακρυσμένη εργασία. Επομένως, καταλαβαίνει κανείς την αναγκαιότητα για μία εφαρμογή ανταλλαγής μηνυμάτων (και δυνητικά δεδομένων) μεταξύ των υπαλλήλων της κάθε εταιρείας, με τρόπο που να προστατεύει τις πληροφορίες της επιχείρησης ενάντια σε κακόβουλες ενέργειες και να παρέχει ασφαλή καταχώρηση τους στην αλυσίδα, αντικαθιστώντας σταδιακά τις παραδοσιακές βάσεις δεδομένων.

Βιβλιογραφία

[1] Nakamoto, S. (2008) *Bitcoin: a Peer-to-Peer Electronic Cash System* [online] Available at: https://bitcoin.org/bitcoin.pdf

[2] Lee, D., T2, B. and Co, S. (1977) *Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups* [online] Available at: https://nakamotoinstitute.org/static/docs/computer-systems-by-mutually-suspicious-groups.pdf

[3] David Chaum [online] Available at: https://en.wikipedia.org/wiki/David_Chaum

[4] Merkle tree [online] Available at: https://en.wikipedia.org/wiki/Merkle_tree

[5] S. Nakamoto [online] Available at: https://en.wikipedia.org/wiki/Satoshi_Nakamoto

[6] Blockchain [online] Available at: https://en.wikipedia.org/wiki/Blockchain#History

[7] Hayes, A. (2022) *Blockchain, Explained* [online] Investopedia. Available at: https://www.investopedia.com/terms/b/blockchain.asp

[8] Iredale, G. (2020) *The History of Blockchain Technology: Must Know Timeline* [online] 101 Blockchains Available at: https://101blockchains.com/history-of-blockchain-timeline/

[9] www.icaew.com (2022) *History of blockchain* [online] Available at: https://www.icaew.com/technical/technology/blockchain-and-cryptocurrency/blockchain-articles/what-is-blockchain/history

[10] Zsigmond, J. (2020) *Creating a Blockchain from Scratch* [online] Medium Available at: https://levelup.gitconnected.com/creating-a-blockchain-from-scratch-9a7b123e1f3e

[11] Anwar, H. (2021) *Blockchain vs Cryptocurrency: Don't Stay Confused!* [online] Available at: https://101blockchains.com/blockchain-vs-cryptocurrency/

[12] GeeksforGeeks (2018) *Blockchain vs Bitcoin* [online] Available at: https://www.geeksforgeeks.org/blockchain-vs-bitcoin/?ref=rp

[13] Frankenfield, J. (2019) *Hash Definition* [online] Available at: https://www.investopedia.com/terms/h/hash.asp

[14] Euromoney (2020) *Blockchain Explained: What is blockchain? | Euromoney Learning* [online] www.euromoney.com Available at: https://www.euromoney.com/learning/blockchain-explained/what-is-blockchain

[15] Massessi, D. (2018) *Blockchain Public/Private Key Cryptography in a nutshell* [online] Medium Available at: https://medium.com/coinmonks/blockchain-public-private-key-cryptography-in-a-nutshell-b7776e475e7c

[16] Cryptopedia Staff (2021*) Public and Private Keys: What Are They?* [online] Available at: https://www.gemini.com/cryptopedia/public-private-keys-cryptography

[17] Frankenfield, J. (2019) *Consensus Mechanism (Cryptocurrency)* [online] Investopedia. Available at: https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp

[18] GeeksforGeeks (2019) *Consensus Algorithms in Blockchain-GeeksforGeeks* [online] Available at: https://www.geeksforgeeks.org/consensus-algorithms-in-blockchain/

[19] Cloud Credential Council (2019) *Knowledge Byte: Understanding Blockchain 101* [online] Available at: https://www.cloudcredential.org/blog/understanding-and-working-with-blockchain-101/

[20] Frankenfield, J. (2019) *Nonce* [online] Available at: https://www.investopedia.com/terms/n/nonce.asp

[21] Frankenfield, J. (2021) *Block Header (Cryptocurrency)* [online] Investopedia Available at: https://www.investopedia.com/terms/b/block-header-cryptocurrency.asp

[22] Hayes, A. (2021) *Target Hash* [online] Investopedia Available at: https://www.investopedia.com/terms/t/target-hash.asp

[23] Hong, E. (2022) *How Does Bitcoin Mining Work?* [online] Available at: https://www.investopedia.com/tech/how-does-bitcoin-mining-work/

[24] Lewis, J. (2019) *Consensus Algorithms Used by Top Blockchain Networks* [online] The Capital Available at: https://medium.com/the-capital/consensus-algorithms-used-by-top-blockchain-networks-b576392cf5ad

[25] Chawla, V. (2020) *What Are The Top Blockchain Consensus Algorithms?* [online] Analytics India Magazine Available at: https://analyticsindiamag.com/blockchain-consensus-algorithms/
[26] GeeksforGeeks (2019) *Consensus Algorithms in Blockchain-GeeksforGeeks* [online] Available at: https://www.geeksforgeeks.org/consensus-algorithms-in-blockchain/
[27] Frankenfield, J. (2019) *Consensus Mechanism (Cryptocurrency)* [online] Investopedia Available at: https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp
[28] Xord Solution (2020) *5 Popular Consensus Algorithms Used In Blockchain* [online] Available at: https://xord.solutions/5-popular-consensus-algorithms-used-in-blockchain/
[29] Chirag (2021) *Analysis of the Blockchain Consensus Algorithms* [online] Available at: https://appinventiv.com/blog/blockchain-consensus-algorithms-guide/
[30] GeeksforGeeks (2019) *Proof of Work (PoW) Consensus* [online] Available at: https://www.geeksforgeeks.org/proof-of-work-pow-consensus/
[31] proofofstake.com (2018) *Proof-of-Stake in Blockchain Technology: All You Need To Know* [online] Available at: https://proofofstake.com/
[32] Ethereum Wiki (2020) [online] Available at: https://eth.wiki/concepts/proof-of-stake-faqs
[33] GeeksforGeeks (2019) *Proof of Stake (PoS) in Blockchain* [online] Available at: https://www.geeksforgeeks.org/proof-of-stake-pos-in-blockchain/?ref=gcse
[34] Frankenfield, J. (2019) *Proof of Stake (PoS)* [online] Investopedia Available at: https://www.investopedia.com/terms/p/proof-stake-pos.asp
[35] Dhar, S. (2019) *What Is Proof-of-Activity (PoA)?* [online] Available at: https://theblockchaincafe.com/what-is-proof-of-activity-poa/
[36] TheLuWizz (2021) *What Is Proof Of Activity* [online] Medium Available at: https://medium.datadriveninvestor.com/what-is-proof-of-activity-1dc176db2131?gi=b9ee06d5cbcc
[37] Shobhit, S. (2021) *Proof of Activity* [online] Investopedia Available at: https://www.investopedia.com/terms/p/proof-activity-cryptocurrency.asp

[38] Hayes, A. (2021) *Proof of Capacity (Cryptocurrency)* [online] Investopedia Available at: https://www.investopedia.com/terms/p/proof-capacity-cryptocurrency.asp

[39] GeeksforGeeks (2021) *Proof of Capacity* [online] Available at: https://www.geeksforgeeks.org/proof-of-capacity/?ref=gcse

[40] Daly, L. (2021) *What Is Byzantine Fault Tolerance?* [online] The Motley Fool Available at: https://www.fool.com/investing/stock-market/market-sectors/financials/cryptocurrency-stocks/byzantine-fault-tolerance/

[41] GeeksforGeeks (2019) *practical Byzantine Fault Tolerance(pBFT)-GeeksforGeeks* [online] Available at: https://www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/

[42] Wouters, O. (2021) *"What's inside a block on the blockchain?"* [online] Medium Available at: https://medium.com/@IDEAL_Stake_Pool/whats-inside-a-block-on-the-blockchain-c148ebe1b28d

[43] Antonopoulos, A.M. and O'reilly Media (2018) *Mastering bitcoin : programming the open blockchain*

[44] Kakarlapudi, P.V. and Mahmoud, Q.H. (2021) *A Systematic Review of Blockchain for Consent Management* Available at: https://doi.org/10.3390/healthcare9020137

[45] CoinMarketCap Alexandria (2021) *Mining Difficulty* [online] Available at: https://coinmarketcap.com/alexandria/glossary/mining-difficulty

[46] Wegrzyn, E.K. and Wang, E. (2021) *Types of Blockchain: Public, Private, or Something in Between* [online] Available at: https://www.foley.com/en/insights/publications/2021/08/types-of-blockchain-public-private-between

[47] GeeksforGeeks (2019) *Types of Blockchain and Chain Terminology* [online] Available at: https://www.geeksforgeeks.org/types-of-blockchain-and-chain-terminology/?ref=rp

[48] Graw, M. (2022) *Best Crypto App 2022 - Top App Revealed* [online] Available at: https://stockapps.com/crypto-apps/

[49] Dilmegani, C. (2022) *Top 17 Blockchain Applications & Use Cases in 2022* [online] Available at: https://research.aimultiple.com/blockchain-applications/

[50] McGuire, A. (2018) *Global Blockchain Adoption: Which Countries are Leading the Charge?-Irish Tech News* [online] Available at: https://irishtechnews.ie/global-blockchain-adoption-which-countries-are-leading-the-charge/

[51] Sharma, T.K. (2021) *Top 10 Countries Leading Blockchain Technology In The World* [online] Available at: https://www.blockchain-council.org/blockchain/top-10-countries-leading-blockchain-technology-in-the-world/

[52] STL Partners (2021) *5 blockchain healthcare use cases* [online] Available at: https://stlpartners.com/articles/digital-health/5-blockchain-healthcare-use-cases/

[53] *Secretum-The blockchain messaging app* [online] Available at: https://secretum.io/

[54] Mulders, M. (2021) *Which Major Banks Have Adopted or Are Adopting the Blockchain?* [online] Blockchain Works Available at: https://blockchain.works-hub.com/learn/Which-Major-Banks-Have-Adopted-or-Are-Adopting-the-Blockchain-

[55] Wikipedia Contributors (2019) *Visual Studio Code* [online] Wikipedia Available at: https://en.wikipedia.org/wiki/Visual_Studio_Code

[56] Wikipedia Contributors (2019) *HTML* [online] Wikipedia Available at: https://en.wikipedia.org/wiki/HTML

[57] Wikipedia Contributors (2019) *JavaScript* [online] Wikipedia Available at: https://en.wikipedia.org/wiki/JavaScript

[58] arimetrics (2021) *What is Framework - Definition, meaning and examples* [online] Available at: https://www.arimetrics.com/en/digital-glossary/framework

[59] MongoDB (2022) *What is the MERN Stack? Introduction & Examples* [online] MongoDB Available at: https://www.mongodb.com/mern-stack

[60] Taylor, D. (2022) *What is MongoDB? Introduction, Architecture, Features & Example* [online] Available at: https://www.guru99.com/what-is-mongodb.html

[61] Codecademy (2022) *What is Express.js?* [online] Available at: https://www.codecademy.com/article/what-is-express-js

[62] Wikipedia Contributors (2019) *React (JavaScript library)* [online] Wikipedia Available at: https://en.wikipedia.org/wiki/React_(JavaScript_library)

[63] Codecademy (2022) *Introduction to JavaScript Runtime Environments* [online] Available at: https://www.codecademy.com/article/introduction-to-javascript-runtime-environments

[64] Wikipedia Contributors (2019*) JavaScript engine* [online] Wikipedia Available at: https://en.wikipedia.org/wiki/JavaScript_engine

[65] Bybit Learn (2021) *What Is Tendermint?* [online] Available at: https://learn.bybit.com/blockchain/tendermint/

[66] Binance Academy. (2021) *Tendermint Explained* [online] Available at: https://academy.binance.com/en/articles/tendermint-explained

[67] docs bigchaindb (2022) *Run BigchainDB with all-in-one Docker-BigchainDB 2.2.2 documentation* [online] Available at: https://docs.bigchaindb.com/en/latest/installation/node-setup/all-in-one-bigchaindb.html

[68] docs bigchaindb (2022) *Properties of BigchainDB-BigchainDB 2.2.2 documentation* [online] Available at: https://docs.bigchaindb.com/en/latest/properties.html

[69] BigchainDB. (2022) *Features & Use Cases • • BigchainDB* [online] Available at: https://www.bigchaindb.com/features/

[70] Gillis, S.A. (2021) *What is a Docker Image and How is it Used?* [online] Available at: https://searchitoperations.techtarget.com/definition/Docker-image

[71] docs bigchaindb (2022) *The WebSocket Event Stream API-BigchainDB Server 1.0.1 documentation* [online] Available at: https://docs.bigchaindb.com/projects/server/en/v1.0.1/websocket-event-stream-api.html

[72] GeeksforGeeks (2019) *ReactJS|Router* [online] Available at: https://www.geeksforgeeks.org/reactjs-router/

[73] Node Package Manager (2022) *react-router-dom* [online] Available at: https://www.npmjs.com/package/react-router-dom

[74] Node Package Manager (2022) *bootstrap* [online] Available at: https://www.npmjs.com/package/bootstrap

[75] Otto, M. (2000) *Bootstrap* [online] Getbootstrap.com Available at: https://getbootstrap.com/

[76] sweetalert2.github.io (2022) *SweetAlert2* [online] Available at: https://sweetalert2.github.io/#usage

[77] Node Package Manager (2022) *sweetalert2* [online] Available at: https://www.npmjs.com/package/sweetalert2

[78] auth0.com (2021) *JWT.IO - JSON Web Tokens Introduction* [online] jwt.io Available at: https://jwt.io/introduction

[79] Node Package Manager (2022) *jwt-then* [online] Available at: https://www.npmjs.com/package/jwt-then

[80] Kolleger, E. (2018) *What is Axios.js and why should I care?* [online] Medium Available at: https://medium.com/@MinimalGhost/what-is-axios-js-and-why-should-i-care-7eb72b111dc0

[81] Node Package Manager (2022) *axios* [online] Available at: https://www.npmjs.com/package/axios

[82] MDN Web Docs (2019) *Promise* [online] Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[83] socket.io (2021) *How it works|Socket.IO* [online] Available at: https://socket.io/docs/v3/how-it-works/

[84] Node Package Manager (2022) *socket.io* [online] Available at: https://www.npmjs.com/package/socket.io

[85] Sookocheff, K. (2019) *How Do Websockets Work?* [online] Kevin Sookocheff Available at: https://sookocheff.com/post/networking/how-do-websockets-work/

[86] Wikipedia Contributors (2022) *WebSocket* [online] Available at: https://en.wikipedia.org/wiki/WebSocket

[87] Engineering Education (EngEd) Program | Section (2021) *Implementing Public Key Cryptography in JavaScript* [online] Available at: https://www.section.io/engineering-education/implementing-public-key-cryptography-in-javascript/

[88] Node Package Manager (2022) *tweetnacl* [online] Available at: https://www.npmjs.com/package/tweetnacl

[89] www.w3schools.com (2022) *React Render HTML* [online] Available at: https://www.w3schools.com/react/react_render.asp

[90] Wikipedia Contributors (2022) *Document Object Model* [online] Available at: https://en.wikipedia.org/wiki/Document_Object_Model

[91] Miell, I. and Aidan Hobson Sayers (2019) *Docker in practice,* Shelter Island, New York: Manning Publications

[92] Amundsen, M. (2016) *Microservice architecture*, O'reilly Media, Inc, Usa

[93] Fontani, D. (2019) *What is Docker?* [online] Medium Available at: https://medium.com/swlh/what-is-docker-28bd2b618eee

[94] TrustRadius Blog (2018) *What are Containers and Why Do You Need Them?* [online] Available at: https:/www.trustradius.com/buyer-blog/what-are-containers-and-why-do-you-need-them

[95] Pittet, S. (2019) *What are containers?* [online] Atlassian Available at: https://www.atlassian.com/continuous-delivery/microservices/containers

[96] Docker (2013) *What is a Container? | Docker* [online] Docker Available at: https://www.docker.com/resources/what-container

[97] Richardson, C. (2017*) Microservices.io* [online] microservices.io Available at: https://microservices.io/

[98] Atchison, L. (2018) *Microservices: What They Are and How They Work* [online] New Relic Blog Available at: https://newrelic.com/blog/best-practices/microservices-what-they-are-why-to-use-them

[99] Node Package Manager (2022) *cors* [online] Available at: https://www.npmjs.com/package/cors

[100] Node Package Manager (2022) *crypto-js* [online] Available at: https://www.npmjs.com/package/crypto-js

[101] Rhodes, D. (2020*) SHA-512 Hashing Algorithm Overview* [online] Available at: https://komodoplatform.com/en/academy/sha-512/

[102] Huang, S. (2018) *Learn & Build a Javascript Blockchain* [online] Medium Available at: https://medium.com/@spenserhuang/learn-build-a-javascript-blockchain-part-1-ca61c285821e

[103] Kurian, A.M. (2018) *How to build a real time chat application in Node.js using Express, Mongoose and Socket.io* [online] Available at:

https://www.freecodecamp.org/news/simple-chat-application-in-node-js-using-express-mongoose-and-socket-io-ee62d94f5804/

[104] Agarwal, H. (2021) *Building a Chat Application From Scratch With Room Functionality* [online] Medium Available at: https://betterprogramming.pub/building-a-chat-application-from-scratch-with-room-functionality-df3d1e4ef662

[105] Nur, I. (2020) *The MERN stack: A complete tutorial* [online] Available at: https://blog.logrocket.com/mern-stack-tutorial/

5. Παράρτημα Κώδικα

5.1 Αρχεία κώδικα εφαρμογής στο front-end

5.1.1 App.js

```
import React from 'react';
import { BrowserRouter, Switch, Route } from 'react-router-dom';
import {Link} from 'react-router-dom';
import Login from './Views/login';
import SignUp from './Views/signup';
import Lobby from './Views/lobby';
import Index from './Views/index';
import Conversation from './Views/conversation';
import IO from 'socket.io-client';
import '../node_modules/bootstrap/dist/css/bootstrap.min.css';
import showMessage from './pop_up_Toasts';

function App() {
  const [socketIO, setSocketIO] = React.useState(null);

  const SocketIO_Setup = () => {

    const userToken = localStorage.getItem('userToken');

     //αν υπάρχει το token χρήστη και δεν έχει πάρει ακόμα κανονική τιμή το
       SocketIO (είναι null δηλαδή και είτε δεν έχει γίνει ακόμα η σύνδεση είτε
       έγινε αποσύνδεση), τότε πρέπει να δημιουργήσω την σύνδεση με το socket
       ώστε να επικοινωνούμε με το backend
    if(userToken && !socketIO) {
      const socket = IO("http://localhost:8000", {
        query: {
            token: localStorage.getItem('userToken'),
        },
      });
       // όταν αποσυνδέεται η σύνδεση socket (πχ για λόγους αποσύνδεσης από το
       internet και όχι χειροκίνητα, δηλαδή με logout) πρέπει να θέσω το state
       του socket, το socketIO σε null
      socket.on('disconnect', () => {
        setSocketIO(null);
      });

       //όταν συνδέεται η σύνδεση socket πρέπει να θέσω το state του socket, το
       socketIO στην τιμή που πήρε από την επιτυχημένη σύνδεση που έγινε
       παραπάνω
      socket.on('connect', () => {
        setSocketIO(socket);
      });
    }
  }

       // την 1η φορά που γίνεται render / εμφανίζεται το τρέχον component App
       (δηλαδή σε όλες τις φάσεις της εφαρμογής), πρέπει να γίνεται η
       επανεκκίνηση του socket για να υπάρχει η απαραίτητη διασύνδεση με το
       backend
  React.useEffect( () => {
    SocketIO_Setup();
  }, []);
```

```
      // εδώ γίνεται η χειροκίνητη αποσύνδεση του χρήστη από την εφαρμογή, ήτοι
      το logout. Αν λοιπόν ο χρήστης πατήσει το κουμπί logout παίρνω τα
      στοιχεία της σύνδεσης socket από το backend, σύμφωνα με το αναγνωριστικό
      token του χρήστη και αν έχει τιμή το state του socketIO (δηλαδή δεν είναι
      null), τότε κάνω την αποσύνδεση με το αντίστοιχο μήνυμα
  const disconnect = () => {
    const socket = IO("http://localhost:8000", {
        query: {
            token: localStorage.getItem('userToken'),
        },
      });
      if(socketIO){
        socket.disconnect();
        localStorage.setItem('userToken', null);
        setSocketIO(null);
        showMessage('success', 'You have logged out.');
      }
  }

  return (
    <BrowserRouter>
      <div className="App">
      <div className="container-fluid">
        <nav className="navbar navbar-expand-sm navbar-light fixed-top">
              <div  className="navbar-brand">      
              Blockchain  Messenger  <i  style={{fontSize:  '12px'}}>by  Gerasimos
              Varnis</i>
            </div>
            {localStorage.getItem('userToken') === 'null' ||
            localStorage.getItem('userToken') === null ?
            <ul className="navbar-nav">
               <li className="nav-item">
                 <Link className="nav-link" to={"/login"}>Login</Link>
               </li>
               <li className="nav-item">
                 <Link className="nav-link" to={"/sign-up"}>Sign-Up</Link>
               </li>
            </ul> :
            <ul className="navbar-nav">
               <li className="nav-item">
                 <Link className="nav-link" to={"/lobby"}>Lobby</Link>
               </li>
               <li className="nav-item">
                 <Link className="nav-link" to={"/login"}
                   onClick={disconnect}>Logout</Link>
               </li>
            </ul>}
                  
        </nav>
      </div>
        <div className="customBox">
          <div className='custombox'>
            <Switch>
              <Route exact path="/login" render={() => <Login
                    SocketIO_Setup={SocketIO_Setup}/>}/>
              <Route exact path='/sign-up' component={SignUp}/>
              <Route exact path='/lobby' render={() => <Lobby
                    socket={socketIO}/>}/>
              <Route exact path='/' component={Index}/>
```

```
            {/* με την : μετά url περνάω ότι είναι μετά την : σαν params από
            αυτό το page στο conversation και έτσι όπως θα δούμε σε εκείνο το
            view μπορώ να πάρω την τιμή των παραμέτρων αυτών μέσω του
            match.params. πρακτικά αυτά εδώ θα είναι αυτά που περνάνε από το
            lobby, όταν πατάω το κουμπί enter σε κάποια συνομιλία */}
            <Route exact path='/conversation/:conversationParameters'
                   render={(props) => <Conversation socket={socketIO}
                   match={props.match}/>}/>
            {/* εδώ γίνεται ο έλεγχος για το αν το url που εισάγει ο χρήστης
            δεν είναι κάποιο από τα προηγούμενα αποδεκτά urls, τα οποία
            αντιστοιχούν και περιγράφουν την λειτουργία της εφαρμογής μας. Αν
            εισάγει κάτι διαφορετικό από αυτό που υπάρχει μέσα στην πρώτη
            παρένθεση του regex, τότε θα γίνεται render ενός μόνο div με
            μήνυμα λάθους, πως η σελίδα δεν βρέθηκε  */}
            <Route exact path={/^\/(?:login|signup|lobby
                   |conversation\/:conversationParameters|)\s*.*[A-Za-
                   z0-9]+$/}
                   render={() =>
                   <div>
                          <br />
                          <h3 style={{textAlign: 'center'}}>Page Not
                          Found!</h3>
                   </div>
                   }/>
          </Switch>
        </div>
      </div>
    </div>
  </BrowserRouter>
 );
}

export default App;
```

5.1.2 login.js

```
import React from 'react'
import axios from 'axios' // αυτο γίνεται για να μπορώ να στείλω requests από το
frontend που είμαι τώρα προς το backend και εκεί γίνεται η διαχείριση ανάλογα με
το userRouting, που ορίσαμε και τις αντίστοιχες μεθόδους (εδώ πχ είναι η μέθοδος
login του userController)
import showMessage from '../pop_up_Toasts';
import { withRouter } from 'react-router-dom';

const Login = (props) => {
      // δημιουργώ τα references, ώστε να υπάρχει δυναμική αναφορά σε
      σχέση με τα πεδία name και password που εισάγει ο χρήστης και έτσι μετά
      από την τιμή/value των inputs αυτών έχω άμεση πρόσβαση με το
      current.value
    const nameReference = React.createRef();
    const passwordReference = React.createRef();

    const loginUser = () => {
        const name = nameReference.current.value;
        const password = passwordReference.current.value;

      // τώρα αφού έχω τις τιμές από αυτά που έχει εισάγει ο χρηστης στα πεδία
      input name, password πρέπει να κάνω την σύνδεση με την βάση και να ψάξω
      τον χρηστη στη mongoDB μέσω του axios.post, που κάνει ένα post request με
```

```
    την μορφή promise και καλεί τον userController, συγκεκριμένα πάει στην
    μέθοδο login. τελικά θα επιστρέψει αποτέλεσμα resolved η rejected. όταν
    επιστραφεί κάτι τότε αυτό θα περιέχεται στην παράμετρο response μέσα στο
    then. αν το axios.post ήταν πετυχημένο δηλαδή έγινε το login, τότε
    δημιουργούμε ένα μηνυματάκι με το makeToast τύπου success και εμφανίζουμε
    από τα δεδομένα του αποτελέσματος το message. Αυτό που περιέχεται μέσα
    στο response είναι αυτό που επιστρέφουμε σαν res json από τον
    userController.
     axios.post('http://localhost:8000/user/login', {
         name,
         password
     }).then(response => {
         showMessage('success', response.data.message);
           // αποθηκεύω προσωρινά το token, το name και το Id του εκάστοτε
           χρηστη που κάνει login στο localStorage (token και Id υπάρχουν στο
           response.data), ώστε μετά να τα χρησιμοποιήσω σε άλλα views με
           το getItem
         localStorage.setItem('userToken', response.data.token);
         localStorage.setItem('userName', name);
         localStorage.setItem('currentUserId', response.data.userId);

            props.history.push('/lobby'); // αφού γίνει το πετυχημένο login
                        κατευθύνω τον χρηστη στο lobby (όπου βλέπει όλους
                        τους χρήστες της εφαρμογής και επιλέγει κάποιον για
                        να συνομιλήσει). δηλαδή το push πρακτικά μετακινεί
                        τον χρήστη από αυτό το view στο lobby
            props.SocketIO_Setup(); // αφού έγινε η πετυχημένη είσοδος στην
                                    εφαρμογή πρέπει να δώσουμε τιμή στο
                                    socket
     }).catch(error => { //αν όμως υπήρξε error κατά το login, μέσω του
                        catch(error) παίρνουμε αυτό το σφάλμα και το
                        εμφανίζουμε πάλι ανάλογα με το throw που έγινε μέσα
                        στον userController.
         if(error.response){
             showMessage('error', error.response.data.message)
         }
     })
 }

 const handleEnter = e => {
     if (e.charCode === 13) {
         loginUser();
       }
 }


 return (
     <div className="login">
         <br />
         <div className="form-group">
             <label htmlFor="name">Name</label>
             <input className="form-control" type="text" name="name"
                 id="name" placeholder='Enter your name'
                 ref={nameReference} onKeyPress={handleEnter}/>
         </div>
         <br />
         <div className="form-group">
             <label htmlFor="password">Password</label>
             <input className="form-control" type="password"
                 name="password" id="password" placeholder='Enter
```

```
                    your password' ref={passwordReference}
                    onKeyPress={handleEnter}/>
            </div>
            <br />
            <div className="button">
                <button className="btn btn-primary btn-block"
                    onClick={loginUser}>Login</button>
            </div>
        </div>
    );
};

export default withRouter(Login); // βάζω το withRouter για να μπορώ να έχω πρόσβαση στην παράμετρο που πέρασα πριν (στο App render όρισα ένα Route για το Login component και πέρασα ως παράμετρο την μέθοδο SocketIO_Setup που υπάρχει εκεί στο App). Έτσι, τώρα μπορώ και χρησιμοποιώ το props.SocketIO_Setup() στο line 27
```

5.1.3 signup.js

```
import React from 'react'
import axios from 'axios' // αυτο γίνεται για να μπορώ να στείλω requests από το frontend που είμαι τώρα προς το backend και εκεί γίνεται η διαχείριση ανάλογα με το userRouting, που ορίσαμε και τις αντίστοιχες μεθόδους (εδώ πχ είναι η μέθοδος sign-up του userController)
import showMessage from '../pop_up_Toasts';
export default function SignUp(props) {
       // δημιουργώ τα references, ώστε να υπάρχει δυναμική αναφορά σε σχέση με τα πεδία name και password που εισάγει ο χρήστης και έτσι μετά από την τιμή/value των inputs αυτών έχω άμεση πρόσβαση με το current.value
    const nameReference = React.createRef();
    const passwordReference = React.createRef();

    const signupUser = () => {
        const name = nameReference.current.value;
        const password = passwordReference.current.value;

        // αφού έχω τις τιμές από αυτά που έχει εισάγει ο χρηστης στα πεδία input name, password πρέπει να κάνω την σύνδεση με την βάση και να γίνει η εγγραφή του χρήστη στη mongoDB μέσω του axios.post. το axios.post κάνει ένα post request με την μορφή promise και καλεί τον userController και συγκεκριμένα την μέθοδο sign-up. τελικά θα επιστρέψει αποτέλεσμα resolved η rejected. όταν επιστραφεί κάτι τότε αυτό θα περιέχεται στην παράμετρο response μέσα στο then. αν το axios.post ήταν πετυχημένο δηλαδή έγινε η εγγραφή του χρήστη στην βάση, τότε δημιουργούμε ένα μηνυματάκι με το makeToast τύπου success και εμφανίζουμε από τα δεδομένα του αποτελέσματος το message. Αυτό που περιέχεται μέσα στο response είναι αυτό που επιστρέφουμε σαν res json από τον userController.
        axios.post('http://localhost:8000/user/sign-up', {
            name,
            password
        }).then(response => {
            showMessage('success', response.data.message);
            props.history.push('./login'); // αφού γίνει το πετυχημένο sign-up κατευθύνω τον χρηστη στο login για να εισάγει τα στοιχεία του και να κάνει είσοδο στην εφαρμογή. δηλαδή το push πρακτικά μετακινεί τον χρήστη από αυτό το view στο login
```

```
        }).catch(error => { // αν όμως υπήρξε error κατά το sign-up του χρήστη,
                            μέσω του catch(error) παίρνουμε αυτό
                            το σφάλμα και το εμφανίζουμε πάλι
                            ανάλογα με το throw που έγινε μέσα στον
                            userController.
            if(error.response){
                showMessage('error', error.response.data.message)
            }
        })
    }

    const handleEnter = e => {
        if (e.charCode === 13) {
            signupUser();
          }
    }

    return (
        <div className="signup">
            <br />
            <div className="form-group">
                <label htmlFor="name">Name</label>
                <input className="form-control" type="text" name="name"
                    id="name" placeholder='Enter your name' ref={nameReference}
                    onKeyPress={handleEnter}/>
            </div>
            <br />
            <div className="form-group">
                <label htmlFor="password">Password</label>
                <input className="form-control" type="password" name="password"
                    id="password" placeholder='Enter your password'
                    ref={passwordReference} onKeyPress={handleEnter}/>
            </div>
            <br />
            <div className="button">
                <button className="btn btn-primary btn-block"
                    onClick={signupUser}>Sign-Up</button>
            </div>
        </div>
    )
}
```

5.1.4 lobby.js

```
import React from 'react'
import axios from 'axios' // αυτο γίνεται για να μπορώ να στείλω requests από το
frontend που είμαι τώρα προς το backend και εκεί γίνεται η διαχείριση ανάλογα με
το userRouting, που ορίσαμε και τις αντίστοιχες μεθόδους (εδώ πχ είναι η μέθοδος
getAllUsers του userController)
import {Link} from 'react-router-dom';
import showMessage from '../pop_up_Toasts';

export default function Lobby(socket) {
    const [users, setUsers] = React.useState([]);
    const reg = /\d+/g;

       // αυτή η μέθοδος είναι μέσα στο useEffect που έχει σαν παράμετρο το [].
       αυτό σημαίνει πως ότι βρίσκεται μέσα στο useEffect θα κληθεί μόνο την
       πρώτη φορά που γίνεται render το component Lobby. Επομένως, όταν ο
```

```
        χρήστης κάνει πετυχημένο login και πάει σε αυτό το view, τότε καλείται
        αυτή η μέθοδος για να φέρει όλους τους χρήστες από την βάση και να τους
        εμφανίσει. Γίνεται ένα get request μέσω του axios.get προς τον σέρβερ
        μας. Σε αυτό το request θέτουμε κάποιους headers και υπάρχει η ιδιότητα
        του Authentication. Αυτό γίνεται, επειδή χρησιμοποιούμε τα jwt - JSON Web
        Tokens για να πιστοποιήσουμε τον χρήστη όταν κάνει το login μέσω ενός
        token που υπογράφεται και ελέγχεται, σύμφωνα με το μοναδικό Id του κάθε
        χρήστη. Ο τρόπος λοιπόν που στέλνουμε το get request εδώ, είναι μέσω
        πιστοποίησης του χρήστη. δηλαδή το bearer authentication ή αλλιώς token
        authentication σημαίνει "δώσε πρόσβαση για request από τον client προς
        τον server, μόνο σε αυτόν που φέρει το token (bearer of token)". Έπειτα,
        το token αυτό ελέγχεται στο backend/server side και αν είναι όντως
        έγκυρο, τότε γίνεται το request.
    const getAllUsers = () => {
        axios.get('http://localhost:8000/user/users', {
            headers: {
                Authorization: 'Bearer ' + localStorage.getItem('userToken'),
            },
        })
        .then( (response) => {
            setUsers(response.data);
              // λοιπόν εδώ με την get παίρνω όλα τους χρήστες που αντιστοιχούν
              ο καθένας σε μία συνομιλία για τον χρηστη με αυτό το συγκεκριμένο
              token, άρα το response.data θα μου επιστρέφει έναν πινακα με όλους
              τους χρήστες (που αργότερα μεταφράζονται σε
              συνομιλίες/conversations. Αν επιλέξω να κάνω enter σε έναν χρήστη
              από την λίστα, τότε μπαίνω στο conversation με αυτό τον χρήστη).
              Έτσι, έχοντας μπεί στο then υπάρχει το αποτέλεσμα της επιτυχημένης
              get και επομένως θέτω το state μου users στο αποτέλεσμα αυτό.
        })
        .catch( (error) => {
             setTimeout(getAllUsers, 2000); // αν υπάρχει error δεν εμφανίζουμε
                                    κάποιο μήνυμα αλλά προσπαθούμε να ξαναφέρουμε
                                    όλους τους χρήστες μετά από 2 δευτερόλεπτα (πχ
                                    ίσως υπήρχε κάποια αδυναμία σύνδεσης με το
                                    backend)
        });
    };

    React.useEffect(() => {
        getAllUsers();

              // αυτό είναι για την περίπτωση που γίνεται refresh στο
              conversation page. επειδή, η σύνδεση socket είναι session και
              χάνεται όταν γίνεται ανανέωση σελίδας. η συνομιλία λειτουργεί με
              τα συγκεκριμένα socket που έχει πάρει ο κάθε χρήστης (από τους 2
              που συνομιλούν), δηλαδή αν ανανεώσω την σελίδα conversations δεν
              ισχύει πλέον το socket που είχα πριν και μου επέτρεπε να μιλήσω με
              τον άλλο χρήστη. για αυτό ανακατευθύνεται ο χρήστης εδώ όταν κάνει
              ένα τέτοιο refresh ώστε να ξαναπατήσει το enter. Ο σκοπός αυτού
του            μηνύματος είναι να ενημερωθεί ο χρήστης αν καταλάθος πατήσει το
              refresh/reload, ότι δεν υπάρχει λόγος να το ξανακάνει διότι η
              εφαρμογή λειτουργεί μια χαρά και realtime όπως και να χει. Επίσης,
              ο δεύτερος έλεγχος για το token γίνεται ώστε να μην βγαίνει αυτό
              το μήνυμα αν έχει αποσυνδεθεί ο χρήστης και πάει στο conversation
              και κάνει refresh.
              if(localStorage.getItem('reloaded') === 'true' &&
                     localStorage.getItem('userToken') !== 'null'){
```

```
                    showMessage('info','You were redirected here to instantiate
                    the socket connection again. There is no need to refresh
                    the page.');
                    localStorage.setItem('reloaded', false);
                }

                //μήνυμα ενημέρωσης του χρήστη για την περίπτωση που το τοπικό
                blockchain μας δεν είναι έγκυρο όταν ο χρήστης κάνει enter στη
                συζήτηση. αυτό εδώ είναι το τελευταίο βήμα. ξεκινάμε από το
                backend, πάμε στο frontend στο αρχείο conversation.js και
                από εκεί καταλήγουμε εδώ.
            if(localStorage.getItem('reconstructedWhenEntering') === 'true'){
                showMessage('info','The blockchain is not valid and has to be
                reconstructed. Please wait for this message to disappear and enter
                again...');
                localStorage.setItem('reconstructedWhenEntering', false);
            }
        }, []);

        return (
            <div>
                {/* εδώ υπάρχει ο έλεγχος για το userToken, αν είναι null. δηλαδή
                πρέπει να έχει γίνει πετυχημένη είσοδος του χρήστη για να μπορέσει
                να δεί την σελίδα αυτή. αν πχ κάποιος πατήσει τη σελίδα χωρίς να
                έχει μπεί στην εφαρμογή ή έχει κάνει logout, τότε θα εμφανίζεται
                μήνυμα ότι δεν έχεις την εξουσιοδότηση για να δεις την σελίδα
                αυτή. αν υπάρχει το token, τότε εμφανίζεται κανονικά. */}
                {localStorage.getItem('userToken') !== 'null' &&
                localStorage.getItem('userToken') !== null ?
                <div>
                    <br />
                    <div className="form-group">
                        <h4 className='text'><i>Click <b><u>Enter</u></b> to
                        start a conversation</i></h4>
                        <hr />
                    </div>
                    <div className='conversations'>
                        <div className='text'>List of Registered Users
                            <br /><br/>
                            {users.filter(u => u.name !==
                            localStorage.getItem('userName')).map(user => (
                                <div key={user._id} className="conversation">
                                    <div>{user.name}</div>
                                    {/* περνάμε σαν παράμετρο στο url της κάθε
                                    συνομιλίας, α) το άθροισμα των αριθμών που
                                    υπάρχουν στα ids των 2 χρηστών που συνομιλούν
                                    ( το οποίο θα είναι πάντα σταθερό), β) το id
                                    αυτού που θα λαμβάνει τα μηνύματα και γ) το
                                    όνομα του. αυτά είναι αναγνωριστικά που
                                    βοηθάνε στην ανταλλαγή των μηνυμάτων με σωστό
                                    τρόπο μετά στο conversation page */ }


                                    <Link to={'/conversation/' +
                                    (parseInt(user._id.match(reg).join('')) +
    parseInt(localStorage.getItem('currentUserId').match(reg).join(''))     +
    '_' + user._id + '_' + user.name}>
                                        <button className="btn btn-primary btn-
                                        block">Enter</button>
                                    </Link>
```

```
                                </div>
                            ))}
                        </div>
                    </div>
                </div>
                :
                <div>
                  <br />
                   <h3 style={{textAlign: 'center'}}>You are not authorized to view
                  this page.<br /><br />Please login or sign-up.</h3>
                </div>
            }
        </div>
    )
}
```

5.1.5 conversation.js

```
import React from 'react';
import { withRouter } from 'react-router-dom';
import showMessage from '../pop_up_Toasts';

const Conversation = ({socket, match}) => {

      // επειδή το socket.io δημιουργεί σύνδεση με τον server με τη μορφή
      session (δηλαδή γίνεται η σύνδεση και ισχύει όσο αυτό το socket instance
      είναι ενεργό) σημαίνει πως αν κάνω refresh μέσα από το conversation view
      γίνεται επανεκκίνηση και υπάρχει άλλο socket instance πλέον, το οποίο
      όμως πρέπει να πάρει τα σωστά στοιχεια για να μπορέσεις να συνομιλήσεις
      με τον χρήστη εκ νέου. Έτσι, αυτό θα γίνεται όταν μπαίνω από το lobby και
      για αυτό αν τυχόν γίνει refresh τότε κάνουμε redirect στο lobby για να
      εισέλθει εκ νέου ο χρηστης στη συζήτηση. Δεν είναι ανάγκη να γίνεται
      refresh, η συνομιλία είναι realtime και γίνεται update όπως πρέπει. Οι
      session συνδέσεις είναι ασφαλότερες, καθώς κάθε φορα υπάρχει διαφορετική
      σύνδεση με αλλα στοιχεια, για αυτό επιλέξαμε το socket. Όμως δεν θέλουμε
      να κάνει re render αυτή η σελίδα, όταν πχ κάνω refresh το lobby page και
      πατάω το enter δε θέλω να μπαίνει σε αυτό το if block και να με
      ξαναγυρνάει στο lobby. Θέλω μόνο όταν πατάω refresh sto conversation page
      (στο τρέχον component δηλαδή) και μηδενίζεται/γίνεται null το socket να
      κάνει το redirect, εξού και ο έλεγχος για το !socket. Διότι, όταν πατάω
      enter μπαίνει στην σελίδα και επειδή κάνει render το react το λαμβάνει
σαν    refresh/reload και αν δεν υπήρχε ο έλεγχος έμπαινε εδώ.
    if (window.performance && !socket) {
        if (window.performance.navigation.type === 1) {
          localStorage.setItem('reloaded', true);
          window.location.href = 'http://localhost:3000/lobby';
        }
      }

      //με βάση το conversationParameters, που όρισα στο App και με τις
      παραμέτρους που βάζω στο Link, όταν πατάει κάποιος το enter κουμπί στο
      lobby για να μπεί σε μία συζήτηση μπορώ και παίρνω τις τιμές που στέλνω

    const conversationID = match.params.conversationParameters.split('_')[0];
      //αυτό είναι το άθροισμα των αριθμών που υπάρχουν στο id του κάθε εκ των
      2 χρηστών. θα είναι πάντα ίδιο είτε μπει ο ένας ως αποστολέας είτε ο
      άλλος. άρα, σύμφωνα με αυτό θα στέλνονται τα μηνύματα και είναι το
      μοναδικό αναγνωριστικό της κάθε συνομιλίας
    const userID = match.params.conversationParameters.split('_')[1];
```

```
    //id του χρήστη που λαμβάνει τα μηνύματα
var receiverName = match.params.conversationParameters.split('_')[2];
    // όνομα του χρήστη που λαμβάνει τα μηνύματα και μπαίνει ως τίτλος στη
    συνομιλία πάνω δεξιά. δηλαδή όταν μπαίνω στην συνομιλία ξέρω ότι αυτός
    που συνομιλώ είναι με το πάνω δεξιά όνομα

const [messages, setMessages] = React.useState([]);
    // το state των μηνυμάτων που υπάρχουν στην συνομιλία

const msgReference = React.useRef();
    // δημιουργώ το reference, ώστε να υπάρχει δυναμική αναφορά σε σχέση με
    το πεδίο του μηνύματος που γράφει ο χρήστης και έτσι έχω άμεση πρόσβαση
    με το current.value

    // στέλνει το μήνυμα και όταν πατάω το enter
const handleEnter = e => {
    if (e.charCode === 13) {
        sendMessage();
      }
}

    //η μέθοδος που καλείται όταν πατάμε send ή enter για να στείλουμε το
    μήνυμα μας. ελέγχουμε αν υπάρχει το socket αρχικά (για να γίνει η σύνδεση
    με το  backend) και κάνω emit (καλώ) την μέθοδο conversationMessage που
    έχει οριστεί στο backend/ αρχείο server.js. Περνάω σαν παραμέτρους το id
    της συνομιλίας, το μήνυμα kai το id του χρήστη που λαμβάνει το μήνυμα.
    Εκεί είναι που εν τέλει αν όλα πάνε καλά καλείται η μέθοδος
    newConvMessage
const sendMessage = () => {
    if(socket){
        socket.emit('conversationMessage', {
            conversationId: conversationID,
            message: msgReference.current.value,
            userId: userID
        });

           //μετά την αποστολή του μηνύματος αρχικοποιώ την αναφορά μου για
          το τι γράφει ο χρήστης, ώστε να είναι έτοιμο για την επόμενη
          αποστολή
        msgReference.current.value = "";
    }
}

    //μόνο την πρώτη φορά που φορτώνει η σελίδα (όταν μπαίνει ο χρήστης πρώτη
    φορά μετά το πάτημα κουμπιού enter)
React.useEffect(() => {
    // αν υπάρχει η σύνδεση socket συνδέεται ο χρήστης στην συζήτηση,
    καλώντας την μέθοδο enterConversation
    console.log(socket);
    if(socket){
        socket.emit('enterConversation', {
            conversationId: conversationID,
            userId: userID
        });

        //αν όταν κάνουμε enter σε μία συζήτηση, εξακριβωθεί από τους
        ελέγχους στο backend ότι το blockchain στην βάση μας δεν είναι
        έγκυρο, τότε γίνεται ανακατεύθυνση στο lobby όπου εμφανίζεται μήνυμα
        ότι ανακατασκευάζεται το blockchain και πρέπει να γίνει enter ξανά.
        socket.on('informTheUserWhenEntering', () => {
```

```
                window.location.href = 'http://localhost:3000/lobby';
                localStorage.setItem('reconstructedWhenEntering', true);
            })

            //φέρνει το ιστορικό όλων των μηνυμάτων από το blockchain της βάσης
            socket.on('getAllMessages', ({allMessages}) => {
                var newMessages = [];
                if(allMessages){
                    for (const msg of allMessages) {
                        const message = {
                            name: msg.username,
                            userId: msg.userSender,
                            message: msg.message
                        }
                        newMessages.push(message);
                    }
                    setMessages(newMessages);
                }
            });
        }

        // όταν φεύγει ο χρήστης από την συνομιλία (κλείνει browser, πάει πίσω,
κάνει refresh), τότε καλούμε την μέθοδο leaveConversation
        return () => {
            if(socket){
                socket.emit('leaveConversation', {
                    conversationId: conversationID,
                })
            }
        }
    },[]);

       //κάθε φορά που αλλάζει το state των μηνυμάτων μπαίνει στην useEffect.
       ΟΜΩΣ, προσοχή. εδώ δεν καλείται η μέθοδος, αλλά αρχικοποιείται/δηλώνεται
       η newConvMessage. Πρακτικά, αυτή θα καλείται όταν γίνεται το emit/η κλήση
       από το backend, μετά την κλήση της παραπάνω μεθόδου conversationMessage.
       Αυτό γίνεται, διότι πριν σταλεί κάποιο μήνυμα υπάρχει διαδικασία στο
       αρχείο server.js (ελέγχεται το blockchain μας, κρυπτογραφείται το μήνυμα,
       γίνεται η συναλλαγή bigchaindb και αποθηκεύεται στο blockchain του
       bigchaindb επιτυχώς και αν όλα αυτά πάνε καλά, τότε ετοιμάζεται η
       εγγραφή/αποθήκευση του μηνύματος/block στην τοπική μας βάση στον πίνακα
       blockchain, αποκρυπτογραφείται το μήνυμα και γίνεται η κλήση της μεθόδου
       newConvMessage (και ερχόμαστε εδώ κάτω) με την παράλληλη ασύγχρονη
       αποθήκευση του block στη βάση)
    React.useEffect(() => {

        if(socket){
            socket.on('newConvMessage', (message) => {
                const newMessages = [...messages, message];
                setMessages(newMessages);
            });

            // αν κατά την διάρκεια της συνομιλίας και συγκεκριμένα όταν πατήσω
            την αποστολή, εξακριβωθεί από τους ελέγχους στο backend ότι το
            blockchain στην βάση μας δεν είναι έγκυρο, τότε βγαίνει μήνυμα
            ενημέρωσης του χρήστη 'να περιμένει όσο το μήνυμα αυτό διαρκεί για
            να γίνει ανακατασκευή του blockchain' περίπου 5 δευτερόλεπτα.
            socket.on('informTheUserWhenSending', () => {
```

```
                showMessage('info','The blockchain is not valid and has to be
                   reconstructed. Please wait for this message to disappear
                   and try again...');
            })
        }
    },[messages])


    return (
        <div>
             {/* εδώ υπάρχει ο έλεγχος για το userToken, αν είναι null. δηλαδή
             πρέπει να έχει γίνει πετυχημένη είσοδος του χρήστη για να μπορέσει
             να δεί την σελίδα αυτή. αν πχ κάποιος πατήσει τη σελίδα χωρίς να
             έχει μπεί στην εφαρμογή ή έχει κάνει logout, τότε θα εμφανίζεται
             μήνυμα ότι δεν έχεις την εξουσιοδότηση για να δεις την σελίδα
             αυτή. αν υπάρχει το token, τότε εμφανίζεται κανονικά. */}
            {localStorage.getItem('userToken') !== 'null' &&
             localStorage.getItem('userToken') !== null ?
                <div>
                    <div className="conversationBox">
                        <div className="receiver">
                            <span>{receiverName}</span>
                        </div>
                        <br />
                        <div className="conversationInterior">
                            {messages.map((msg, index) => (
                                <div key={index} className="message">
                                    <span className={receiverName !== msg.name ?
                                        "me"   :   "others"}>{receiverName   !==
                                        msg.name  ?  'Me'  :  msg.name}:  </span>
                                        {msg.message}
                                </div>
                            ))}
                        </div>
                        <div className="conversationHandlers">
                            <div>
                                <input type="text" name='message'
                                  placeholder='Type  here...'  ref={msgReference}
                                  onKeyPress={handleEnter} />
                            </div>
                            <div>
                                  <button className="btn btn-primary btn-block"
                                  onClick={sendMessage}>Send</button>
                            </div>
                        </div>
                    </div>
                </div>
                :
                <div>
                    <br />
                     <h3 style={{textAlign: 'center'}}>You are not authorized to
                           view this page.<br /><br />Please login or sign-
                           up.
                     </h3>
                </div>
            }
        </div>
    );
};
```

```
export default withRouter(Conversation);
// βάζω το withRouter για να μπορώ να έχω πρόσβαση στις παραμέτρους που πέρασα πριν (στο App render όρισα ένα Route για το Conversation component, αρχικά με την ενσωμάτωση του conversationParameters και πέρασα ως παράμετρους το socket kai to match). Έτσι,τώρα μπορώ και χρησιμοποιώ και το socket και το match.params για να πάρω τα αναγνωριστικά της συνομιλίας.
```

5.2 Αρχεία κώδικα εφαρμογής στο back-end

5.2.1 app.js

```
const express = require('express');

//η εφαρμογή μας θα είναι express εφαρμογή, δηλαδή χρησιμοποιούμε το express σαν την βάση πάνω στην οποία χτίζεται η εφαρμογή μας και μπορούμε να χρησιμοποιήσουμε όλες τις συναρτήσεις του για να κάνουμε την ενδιάμεση σύνδεση μεταξύ front και back end. Πρακτικά και με απλά λόγια, ορίζουμε πως θα διαχειρίζονται τα requests από το front στο back. Είναι μία απλή και χρήσιμη τεχνική καθώς χρησιμοποιώντας τις μεθόδους του express έχουμε ανά πάσα στιγμή πρόσβαση σε όλα τα requests μέσω των αντικειμένων αιτήματος (req), απόκρισης (res) και στην επόμενη συνάρτηση next που διαχειρίζεται αυτόματα το τι θα γίνει (αν πχ δεν κάνουμε κάτι εμείς πιο πριν ή δεν καλύπτονται οι συνθήκες)
const app = express();

//γενικά με το app.use ενσωματώνω στην εφαρμογή μου διάφορα χαρακτηριστικά και επιλογές που μου δίνονται από το express, όπως βλέπουμε παρακάτω

//ενσωματωμένη λειτουργία ενδιάμεσου λογισμικού στο Express. Αναλύει τα εισερχόμενα requests ως JSON και η όλη διαχείριση γίνεται πάνω σε αυτό το πρότυπο
app.use(express.json());
// με απλά λόγια αυτό χρησιμοποιείται για να μπορούμε να περνάμε περιεχόμενο στα αιτήματα εμφωλευμένα/nested αντικείμενα μέσα σε άλλα αντικείμενα κοκ, αλλιώς δεν γίνεται κάτι τέτοιο αν δεν το είχαμε ορίσει.
app.use(express.urlencoded({extended: true}));
//Το "CORS" σημαίνει Cross-Origin Resource Sharing. επιτρέπει να γίνονται requests από την εφαρμογή μας σε έναν άλλο ιστότοπο πχ σε κάποια APIs που χρησιμοποιούμε. Ουσιαστικά παρακάμπτει τις ρυθμίσεις που έχουν κάποιοι browsers και οι οποίες το απαγορεύουν σύμφωνα με την πολιτική Same origin Policy (SOP). Και για να μην υπάρχουν θέματα σε διαφορετικούς browsers κλπ το ενεργοποιούμε global στην εφαρμογή μας, αφού πρώτα εγκαταστήσαμε το package cors με το npm install --save cors
app.use(require('cors')());

// πρέπει να ενημερώσω την εφαρμογή μου πως θα χρησιμοποιηθεί για τις λειτουργίες του χρήστη το αρχείο userRouting, στο οποίο ορίζεται τι γίνεται οταν ο χρήστης κανει requests στον σερβερ. Δηλαδή πχ όταν ο χρήστης πατάει το κουμπί login τότε γίνεται από το frontend μέσω του axios post request sto http://localhost:8000/user/login και σε αυτή την περίπτωση πρέπει σύμφωνα με το αρχείο userRouting να κληθεί η μέθοδος login από τον userController. Ομοίως με το sign up κουμπί και με την εμφάνιση όλων των χρηστών (από getAllUsers) όταν μπαίνει ο χρήστης στο lobby.

app.use('/user', require('./userRouting'));

// εδώ αρχικά κάνω την σύνδεση με το αρχείο handlingErrors, στο οποίο υπάρχουν έλεγχοι για διάφορα σφάλματα και το τί γίνεται για να διαχειριστούν. Ενημερώνω την εφαρμογή μου ότι για τέτοια σφάλματα θα αναλαμβάνει το αρχείο αυτό και τα χειρίζεται.
```

```
const errors = require('./handlingErrors');
app.use(errors.mongodbErrors);
module.exports = app;
```

5.2.2 User.js

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
    name: {
        type: String,
        required: 'Name is required'
    },

    password: {
        type: String,
        required: 'Password is required'
    },

    address_publicKey: {
        type: String,
        required: 'address_publicKey is required'
    }
},
  {timestamps: true}
);

// εδώ κάνω export το model της βάσης User, ώστε να μπορώ να έχω πρόσβαση μετά και να κάνω διάφορες ενέργειες όπως πχ να βρώ χρήστη μέ την find
module.exports = mongoose.model('User', userSchema);
```

5.2.3 Blockchain.js

```
const mongoose = require('mongoose');

//εδώ έχουμε τον πίνακα blockchain, όπου κάθε εγγραφή θα είναι και 1 block. Επομένως, οι στήλες αυτού του πίνακα πρέπει να έχουν τις ιδιότητες, το τι θα περιέχει το κάθε block, δηλαδή το message, το timestamp (για το πότε δημιουργήθηκε το block ή αντίστοιχα για το πότε ενσωματώθηκε επιτυχώς η συναλλαγή στο blockchain του bigchaindb), το previous block hash και το current block hash, το index για να ξέρουμε με ποιά σειρά μπήκαν τα blocks (και να τα φέρνουμε με τον ίδιο τρόπο όταν γίνετα η ανακατασκευή), το name, που είναι κοινό για όλα τα blocks (περιγράφει το όνομα του blockchain μας και μέσω αυτού γίνεται το fetch από το bigchaindb) και τα ids του αποστολέα και του παραλήπτη.  Γίνεται η παραδοχή ότι κάθε block θα είναι μόνο ένα μήνυμα, ένα message transaction και όχι παραπάνω.
const blockchainSchema = new mongoose.Schema({
    message: {
        type: String
    },

    senderId: {
        type: String,
        ref: 'User'
    },
```

```
    recipientId: {
        type: String,
        ref: 'User'
    },

    timestamp: {
        type: String,
        required: ' is required',
    },

    previousBlockHash: {
        type: String,
    },

    currentBlockHash: {
        type: String,
    },

    index: {
        type: Number
    },

    name: {
        type:String
    }
});

//  εδώ κάνω export το model της βάσης Blockchain, ώστε να μπορώ να έχω πρόσβαση μετά και να κάνω διάφορες ενέργειες όπως πχ find, delete κλπ
module.exports = mongoose.model('Blockchain', blockchainSchema);
```

5.2.4 userRouting.js

```
const router = require('express').Router();// που θα κατευθύνονται τα requests ανάλογα με το url που περιέχουν. πχ βλέπουμε παρακάτω πως όταν γίνεται request στο http://localhost:3000/login στο frontend αυτό αντιστοιχεί στο http://localhost:8000/user/login στο backend και τότε καλείται η μέθοδος login από τον userController.

const {appErrors} = require('./handlingErrors')

const userController = require('./Controllers/userController');

const authentication = require('./authentication');

// αυτό που θέλουμε εδώ είναι όταν γίνονται τα requests για login, sign-up και getAllUsers να ελέγχουμε και να δεσμεύουμε τα errors που προκύπτουν από την μεθόδους αυτές του userController. Επίσης, για την μέθοδο getAllUsers συμπεριλαμβάνω και το authentication στο ενδιάμεσο, δηλαδή θα πρέπει να ελέγχεται πως το token από τους headers του request είναι έγκυρο και κατ'επέκταση ο χρήστης είναι έγκυρος, είναι αυτός που όντως έκανε το request, άρα έχει το δικαίωμα να δεί όλους τους χρήστες, να δει το αποτέλεσμα που θα επιστρέψει η μέθοδος αυτή.
router.post('/login', appErrors(userController.login));
router.post('/sign-up', appErrors(userController.signup));
router.get('/users', authentication, appErrors(userController.getAllUsers));
module.exports = router;
```

5.2.5 handlingErrors.js

```
//Αυτά είναι ελάχιστα τροποποιημένα από το documentation για το error handling στο express.js https://expressjs.com/en/guide/error-handling.html. Δεν θα μπορούσαν να γίνουν και τα 2 σε ένα export, διότι το 1ο δέχεται μία μέθοδο ως παράμετρο και μετά την τρέχει για να 'πιαστεί' το τυχόν σφάλμα, ενώ το 2ο τρέχει σε κάθε σφάλμα που βγαίνει από requests στη βάση mongodb

//Σφάλματα εφαρμογής
//εδώ περνάμε αρχικά σαν παράμετρο fn κάποια μέθοδο του userController, όπως γίνεται στο userRouting. Αφού λοιπόν περάσουμε την μέθοδο τότε επιστρέφουμε μία συνάρτηση με 3 παραμέτρους request (που είναι το http request), το res (που είναι η απάντηση) και το next (που είναι η προκαθορισμένη default τεχνική/μέθοδος του express.js framework για να διαχειρίζονται αυτόματα τα σφάλματα, χωρίς την επέμβαση του χρήστη). Και μέσα σε αυτή την συνάρτηση γίνεται η διαχείριση των σφαλμάτων.
exports.appErrors = (controllerMethod) => {
  return function (req, res, next) {
    //Όταν καλείται κάποια από τις 3 μεθόδους του userController και προκύψει error τότε τρέχει το παρακάτω. το σφάλμα 'πιάνεται' με το catch και αν είναι μορφής string επιστρέφεται το μήνυμα σφάλματος σαν response json αλλιώς διαχειρίζεται με την default μέθοδο next του express
    controllerMethod(req, res, next).catch((err) => {
      console.log(err);
      if (typeof err === "string") {
        res.status(400).json({
          message: err,
        });
      } else {
        next(err);
      }
    });
  };
};

//Σφάλματα βάσης δεδομένων mongodb
//περνάμε σαν παραμέτρους το err (που είναι το σφάλμα που θα προκύψει από κάποιο http request που φτάνει εν τέλει στη βάση μας), το request (που είναι το http request), το res (που είναι η απάντηση) και το next (που είναι η προκαθορισμένη default τεχνική/μέθοδος του express.js framework για να διαχειρίζονται αυτόματα τα σφάλματα, χωρίς την επέμβαση του χρήστη)
exports.mongodbErrors = (err, req, res, next) => {
  //αν το err δεν περιέχει κάποιο πεδίο/object errors (δηλαδή δεν είναι ξεκάθαρο ποιό ήταν το σφάλμα), τότε το περνάμε στην next μέθοδο και το διαχειρίζεται αυτή
  if (!err.errors) return next(err);
  //αλλιώς, βρίσκουμε τα keys του αντικειμένου errors, δηλαδή τα ονόματα των ιδιοτήτων του (που θα είναι 1 ή περισσότερα μηνύματα σφαλμάτων). αυτό επιστρέφεται με την μορφή πίνακα και μετά μπορούμε να έχουμε πρόσβαση στις τιμές κάνοντας ένα loop στον πίνακα αυτόν, όπως παρακάτω
  const errorKeys = Object.keys(err.errors);
  let message = "";
  //άρα τώρα για τον πίνακα αυτό των σφαλμάτων, θα πάρουμε το κάθε ένα από αυτά και θα δημιουργήσουμε ένα string με τα μηνύματα που περιέχουν
  errorKeys.forEach((key) => (message += err.errors[key].message + ", "));
  // παίρνω το string αφού πρώτα το κόψω, αφαιρώντας το τελευταίο ',' για αυτό βάζουμε το length - 2
  message = message.substr(0, message.length - 2);

  // και τελικά επιστρέφουμε το μήνυμα ως json που περιέχει όλα τα μηνύματα των σφαλμάτων, προσδιορίζοντας τον τύπο του response ως 400 / bad request και σταματάει η διαδικασία αυτή του request
```

```
  res.status(400).json({
    message,
  });
};
```

5.2.6 authentication.js

```
const jwt = require('jwt-then');

module.exports = async (req, res, next) => {
    try{
        // αν δεν υπάρχει το token που δίνεται στον χρήστη όταν κάνει πετυχημένο
        login, τότε πρέπει να βγαίνει error
        if(!req.headers.authorization){
            throw 'You are not authorized / User token is missing'
        }

        //παίρνω το token του χρήστη που έχει κάνει login από το request και
        συγκεκριμένα από τους headers και το κάνω split στο space, δηλαδή θα
        πάρω έναν πίνακα που θα έχει ['something', '985fhashvc772fa75742jdvn',
        ...]. άρα, για αυτό παίρνω το δεύτερο στοιχείο.
        const token = req.headers.authorization.split(' ')[1];

        //τώρα πρέπει να κάνω verify το token και να περιμένω να γίνει το verify
        για να συνεχίσω πιο κάτω. Όταν στο login κάνω sign για το token
        στον userController, χρησιμοποιώ το id του user (αυτό δηλαδή που
        δίνει αυτόματα το mongodb σε κάθε εγγραφη). αν το token που πήρα
        από τους headers δεν κάνει επιτυχώς το verify τότε θα βγεί error
        και πάμε παρακάτω στο catch. αλλιώς αν πάνε όλα καλά και μου
        επιστρέψει το json web token τότε συνεχίζω
        const payload = await jwt.verify(token, process.env.SECRET);

        req.payload = payload;// άρα, εδώ θέτω το payload του request ως αυτό
                              που πήρα σαν απάντηση από το verification του
                              token παραπάνω.
        next();
    }
       //401 status για not authorized, δηλαδή ότι δεν είναι έγκυρο το token με
       το οποίο έγινε το request για login ή για να φέρουμε όλους τους
       χρήστες μέσω της getAllUsers()
    catch(error) {
        res.status(401).json({
            message: 'Not Authorized'
        })
    }
}
```

5.2.7 userController.js

```
const mongoose = require('mongoose');
const User = mongoose.model('User');
const jwt = require('jwt-then');
const nacl = require('tweetnacl');
nacl.util = require('tweetnacl-util');
const sha512 = require('crypto-js/sha512');
const fromUint8ArraytoHexadecimalString = uint8Array =>
  uint8Array.reduce((string, byte) =>
              string + byte.toString(16).padStart(2,    '0'), '');
```

```
//πρόκειται για μεθόδους που επιστρέφουν promises, οπότε θα γίνουν με την μορφή
async/await συναρτήσεων
exports.login = async (req, res) => {

    const {name, password} = req.body;
    // βρίσκω τον χρηστη στη βάση με το name του
    const userExists = await User.findOne({name: name});

    // αν δε βρεθεί ο χρηστης τότε θα βγαίνει error και ενημερώνεται ο χρήστης
    if(!userExists) {
        throw 'User does not exist or the username is not correct.'
    }

     // πρέπει να ελέγχω αν ο χρήστης έχει βάλει το σωστό password, δηλαδή το
       ίδιο με αυτό που υπάρχει στη βάση και το οποίο έβαλε όταν έκανε το sign
       up
    if(sha512(JSON.stringify(password)).toString() !== userExists.password){
        throw 'The password is not correct.'
    }

    const token = await jwt.sign({id: userExists.id}, process.env.SECRET);



    res.json({
        message: 'User "' + name + '" has logged in.',
        token: token,
        userId: userExists._id
    })
}

exports.signup = async (req, res) => {
    const {name, password} = req.body;

    const userExists = await User.findOne({name: name});

    if(name === ''){
        throw 'Name is required';
    }

      // αν ο χρήστης υπάρχει ήδη με αυτό το όνομα θα βγαίνει error και
       ενημερώνεται ο χρήστης
    if(userExists) {
        throw 'This user already exists. Please try a different name.';
    }

     // αν το password δεν είναι τουλάχιστον 8 χαρακτήρες θα βγαίνει error και
       ενημερώνεται ο χρήστης
    if(password.length < 8){
        throw 'Password must contain at least 8 characters.';
    }

    const keyPair = nacl.box.keyPair();
    const publicKey = keyPair.publicKey;

    const hexPublicKey = fromUint8ArraytoHexadecimalString(publicKey);

    const user = new User({
        name,
```

```
        password: sha512(JSON.stringify(password)).toString(),
        address_publicKey: hexPublicKey
    });

    await user.save();

     // Τώρα αφού έγινε η αποθήκευση του χρήστη με τα παρπάνω 3 στοιχεία στη
       βάση, πρέπει να επιστρέφω ένα promise με τη μορφή μηνύματος json
       προκειμένου να ενημερώνεται ο χρήστης ότι η εγγραφή του ήταν επιτυχημένη.
       αν δεν είναι πετυχημένη βγαίνει κάποιο μήνυμα από τα παραπάνω 2 if blocks
    res.json({
        message: 'User "' + name + '" has signed-up.'
    });
}

//βρίσκει όλους τους χρήστες από τη βάση και τους επιστρέφει με json
exports.getAllUsers = async (req,res) => {
    const users = await User.find();
    res.json(users);
}
```

5.2.8 server.js

```
require('dotenv').config() // φορτώνω μέσω του dotenv πακέτου το αρχείο env,
ώστε να συμπεριληφθούν οι σταθερές που έχω ορίσει στο αρχείο μου ώς μεταβλητές
της τρέχουσας διεργασίας και να έχω πρόσβαση σε αυτές από το process.env. Πχ
χρησιμοποιώ την σταθερά process.env.DATABASE που έχω ορίσει μέσα στο αρχείο env
για να κάνω τη σύνδεση με το mongoose.

//SET UP THE DATABASE
const mongoose = require('mongoose');

const nacl = require('tweetnacl');
nacl.util = require('tweetnacl-util');
const sha512 = require('crypto-js/sha512');

// έτσι γίνονται οι συναλλαγές στο blockchain bigchaindb και με το searchAssets
τις παίρνω πίσω. άρα, κάθε φορα που στέλνω ένα μήνυμα θα το κάνω με
κρυπτογράφηση το message. κάθε φορα στο getAllMessages τα φέρνω από το bigchain
ψαχνοντας τα blocks που έχουν σαν αποστολέα η/και παραλήπτη τους 2 χρηστες που
αλληλεπιδρούν στο κάθε conversation.

const driver = require('bigchaindb-driver');
const API_PATH = 'http://localhost:9984/api/v1/'; // έχοντας τρέξει το docker
container gia to bigchaindb έχουν ρυθμιστεί τα πάντα στο σύστημα μου, ώστε να
δημιουργώ ένα node του bigchaindb network μέσω του τοπικού μου δικτύου και οι
συναλλαγές μου να επικυρώνονται μέσω 2-3 validator nodes που είναι πάλι στο
local δίκτυο μου. Επομένως, αφ'ενός προσφέρουμε στο συνολικό bigchain blockchain
επιπλέον κόμβους κάνοντας το πιο ισχυρό και  αφ'ετέρου έχουμε έναν τρόπο να
είμαστε πιο ασφαλείς, αφού όλα τα transactions μας εκτελούνται και επικυρώνονται
μέσω του localhost μας (χρησιμοποιώντας το consensus της Tendermint, του οποίου
ο αλγόριθμος έχει ενσωματωθεί όταν τρέξαμε το container bigchaindb) πριν πάνε
στα blocks κατανεμημένου δικτύου της bigchaindb.
const conn = new driver.Connection(API_PATH);
//const conn = new driver.Connection('https://test.ipdb.io/api/v1/'); // αυτό
είναι το testnet της bigchaindb και κάθε μέρα κάνει reset τα transactions για
λόγους GDPR. Χρησιμοποιήθηκε μόνο στην αρχή για να δω αν και πως δουλεύει όλο
αυτό
```

```
//συνδέομαι στην τοπική βάση mongodb μέσω της σταθεράς από το αρχείο env,δηλαδή κάνω την σύνδεση για το περιβάλλον mongodb://localhost/MERNBlockchainMessenger
mongoose.connect(process.env.DATABASE , {
    useUnifiedTopology: true,
    useNewUrlParser: true
});

//ενημερώνει όταν βρεθεί error στη σύνδεση με την τοπική βάση μας και έτσι ξέρουμε ποιό είναι αυτό το error.
mongoose.connection.on('error', (error) => {
    console.log("There is an error with the mongoose connection: " +
              error.message);
});

// ενημερώνει την πρώτη φορά μόνο, με το που γίνει πετυχημένη σύνδεση στην τοπική βάση μας.
mongoose.connection.once('open', () => {
    console.log('Mongo database is connected')
})

//μετά την επιτυχημένη σύνδεση στη βάση πρέπει να γίνει η διασύνδεση με τα μοντέλα που έχω στον κώδικα μου. τα models πρακτικά είναι ο κάθε πινακας που θα υπάρχει στη βάση. 2 models, άρα 2 πίνακες στην τοπική βάσης μας
require('./Models/User');
require('./Models/Blockchain');

const app = require('./app');

const Server = app.listen(8000, () => {
    console.log('Server is listening on port 8000');
});

//δημιουργώ το middleware μέσω του socket για τον server μου. κάνει την σύνδεση με τον σέρβερ μας, δηλαδή το πως θα γίνεται η επικοινωνία του backend με το frontend
const SocketIO = require("socket.io")(Server, {
    allowEIO3: true,
    cors: {
      origin: true,
      methods: ['GET', 'POST'],
      credentials: true
    }
});

const jwt = require('jwt-then');

const User = mongoose.model('User');
const Blockchain = mongoose.model('Blockchain');

//**********ΜΕΘΟΔΟΙ/ΣΥΝΑΡΤΗΣΕΙΣ BLOCKCHAIN ΚΑΙ ΣΥΝΔΕΣΙΜΟΤΗΤΑ

//Γενικά για τις παρακάτω 2 μεθόδους πρέπει να ξέρουμε πως ο πίνακας Uint8Array είναι ένας πίνακας με στοιχεία ακεραίους 8 bit, που πρακτικά θα είναι αριθμοί bytes από 0 ως 255 και είναι η μορφή που πρέπει να δέχονται οι μέθοδοι που κάνουν generate τα κλειδιά για την κρυπτογράφηση/απόκρυπτογράφηση των μηνυμάτων. Οπότε θα δούμε παρακάτω διάφορες μετατροπές από δεκαεξαδικά strings σε Uint8Array και αντίστροφα.

const fromHexadecimalStringtoUint8Array = hexadecimalString =>
```

```
// αρχικά με το regex βρίσκει όλους τους χαρακτήρες (η τελεία '.' σημαίνει όλοι οι χαρακτήρες εκτός τις αλλαγές γραμμής \n κλπ) που έχουν 1 ή 2 μήκος, διότι ξέρουμε πως οι δεκαεξαδικοί έχουν μήκος 1 ή 2 (χωρίς τα 0x στην αρχή κλπ). άρα βρίσκει όλους τους δεκαεξαδικούς σε έναν πίνακα και μετά για όλο τον πίνακα βρίσκει για κάθε στοιχείο το αντίστοιχο σε byte από 0 ως 255 και το καταχωρεί σαν στοιχείο του Uint8Array.
   new Uint8Array(hexadecimalString.match(/.{1,2}/g).map(byte => parseInt(byte, 16)));

const fromUint8ArraytoHexadecimalString = uint8Array =>
// αντίστοιχα εδώ παίρνει τα στοιχεία του Uint8Array που είναι αριθμοί από το 0 ως το 255 δηλαδή bytes και σχηματίζει σε κάθε επανάληψη το δεκαεξαδικό string. αυτό γίνεται με το reduce που λειτουργεί προσθετικά. δηλαδή το str είναι αυτό που θα πάρουμε τελικά σαν αποτέλεσμα και σε κάθε επανάληψη περιέχει το άθροισμα όλων των προηγούμενων στοιχείων του πίνακα αφού πρώτα μετατράπηκαν σε δεκαεξαδικά, το byte είναι το εκάστοτε στοιχείο του πίνακα Uint8Array που προστίθεται στο προηγούμενο άθροισμα και το '' είναι η αρχική τιμή για το str. Δηλαδή ξεκινάμε από το str = '' ένα κενό string, έπειτα στην 1η επανάληψη προσθέτουμε στο str το δεκαεξαδικό αντίστοιχο του 1ου byte στοιχείου του πίνακα Uint8Array, στην 2η επανάληψη προσθέτουμε στο str το δεκαεξαδικό αντίστοιχο του 2ου byte στοιχείου κοκ μέχρι να τελειώσει το loop. Τελικά, θα έχουμε το δεκαεξαδικό string που αντιστοιχεί στον πίνακα Uint8Array που περάσαμε σαν παράμετρο με το bytes. το padStart γίνεται για να προκύπτει κάθε φορά δεκαεξαδικός με μήκος 2 (αν έχει μήκος 1, τότε προσθέτει το 0 στην αρχή), καθώς στην γλώσσα προγραμματισμού javascript πρέπει κάθε byte όταν μετατρέπεται σε δεκαεξαδικό να αντιπροσωπεύεται με 2 ψηφία, ανεξάρτητα αν κανονικά στο 16αδικό μπορεί να έχει ένα ψηφίο (πχ το 0 θα πρέπει να είναι εδώ 00), προκειμένου να δουλέψει αυτή η μετατροπή.
  uint8Array.reduce((string, byte) =>
              string + byte.toString(16).padStart(2,'0'), ''
  );

const GetDateTimeNow = () => {
    var DateTime = new Date();
    var date = DateTime.getFullYear()+ '/' + (DateTime.getMonth() + 1) + '/' +
               DateTime.getDate();
    var time = DateTime.getHours() + ":" + DateTime.getMinutes() + ":" +
               DateTime.getSeconds();
    return date + ' ' + time;
}

const ComputeTheHashOfThisBlock = (MessageTransactions, PreviousBlockHash,
                           Timestamp, SenderId, RecipientId, Index) => {
     return  sha512(JSON.stringify(MessageTransactions)  +  PreviousBlockHash  +
                    Timestamp + SenderId + RecipientId + Index).toString();
}

const VerifyTheBlockchain = async(blockchain) => {
    var validity = true;

    // περιμένω να πάρω την απάντηση από το promise searchAssets για το αντικείμενο που περιέχει το blockchain στο BigchainDB
    const res = await conn.searchAssets('BlockchainMessenger_v1');

       // αν εχουν διαγραφει τα παντα και δεν υπάρχει τίποτα
    if(blockchain.length === 0) {
        validity = false;
    }
    //η αν έχει διαγραφεί το genesis block
    else if(blockchain.length > 0){
```

```
        if(blockchain[0].index !== 0){
            validity = false;
        }
    }

       // από το i=1, γιατί εδώ δεν μας ενδιαφέρει το genesis block (το ελέγξαμε
       παραπάνω). Έτσι, ελέγχω μονο αν η αλυσίδα έχει πάνω από 2 στοιχεια δηλαδή
       το length είναι >= 2. Αν έχει μονο το genesis δεν χρειάζεται να ελέγχουμε
       ξανά
    if(blockchain.length>=2){
        for(var i=1; i<blockchain.length; i++){

            var ThisBlock = blockchain[i];
            var PreviousBlock = blockchain[i-1];

              // εδώ γίνεται ο έλεγχος αν είναι έγκυρο το blockchain, με την
                    διασταύρωση των hashes
              if( ThisBlock.previousBlockHash !== PreviousBlock.currentBlockHash
                                ||
                ThisBlock.currentBlockHash !==
              ComputeTheHashOfThisBlock(ThisBlock.message,
       ThisBlock.previousBlockHash,ThisBlock.timestamp,ThisBlock.senderId,
       ThisBlock.recipientId, ThisBlock.index) ){
                validity = false;
                break;
            }
        }
    }

       // αυτό λοιπόν πρόκειται για ένα αντικείμενο promise, αν όλα πήγαν καλά
       και περιέχει δεδομένα (δηλαδή το blockchain μου) και μέχρι στιγμής το
       blockchain είναι έγκυρο, τότε πρέπει να ελέγχω αν το μήκος από το
       response (τον πίνακα blockchain, που είναι όμως στο bigchaindb και είναι
       αξιόπιστο) είναι ίσο με το μήκος του blockchain μας στην τοπική βάση. Αν
       δεν είναι ίδιο, σημαίνει ότι έχει διαγραφεί κάποιο block από την
       τοπική βάση και πρέπει να γίνει ανακατασκευή, άρα η εγκυρότητα να πάει
       στο false
    if(res && validity){
        if(res.length !== blockchain.length){
            validity = false;
        }
    }

    console.log(validity);
    return validity;
}

const ReconstructTheBlockchain = async() => {

       //πρέπει να ξαναφέρω από το bigchaindb όλες τις συναλλαγές που
       συγκεντρωτικά δημιουργούν το τοπικό μου blockchain. Πρακτικά, κάθε
       συναλλαγή στο bigchaindb είναι και ένα block στο τοπικό blockchain μου.
       οπότε διαγράφω τα πάντα που εχουν απομείνει και θα το φέρω ξανά με την
       σωστή του μορφή
    const removed = await Blockchain.deleteMany({});

        // η τρέχουσα μέθοδος καλείται μόνο όταν έχει αποτύχει ο έλεγχος
       εγκυρότητας. Δηλαδή είτε δεν υπάρχει τίποτα στην τοπική βάση μας στον
       πίνακα blockchain (διαγράφηκαν όλα τα blocks και το removed.deletedCount
       πρακτικά είναι 0) είτε διαγράφηκαν όλα τα εναπομείναντα blocks
```

```
    if(removed.deletedCount>=0){
        conn.searchAssets('BlockchainMessenger_v1').then( response => {
            for(let i=0; i<response.length;i++){
                let blockdata = response[i].data;

                const block = new Blockchain({
                    message: blockdata.message,
                    senderId: blockdata.senderId,
                    recipientId: blockdata.recipientId,
                    timestamp: blockdata.timestamp,
                    previousBlockHash: blockdata.previousBlockHash,
                    currentBlockHash: blockdata.currentBlockHash,
                    index: blockdata.index,
                    name: blockdata.name
                })
                block.save();
            }
        })
    }
}

// αυτή θα καλείται μια φορά όταν πχ δημιουργώ πρώτη φορα το blockchain μου με
το συγκεκριμένο όνομα (που σημαίνει ότι στη βάση δεν υπάρχει τίποτα στον πινακα
blockchain), μετά θα υπάρχει μόνιμα όλο το blockchain μας στη bigchaindb και
όποτε εντοπίζεται άκυρο blockchain τότε θα το φέρνω από εκεί και θα γίνεται
reconstruct με την παραπάνω μέθοδο
const CreateGenesisBlock = async(idS) => {
     const userS = await User.findOne({_id: idS}); //βρίσκω τον χρήστη που
       ενεργεί ως αποστολέας

     //δημιουργώ με βάση το δημόσιο κλειδί που έχει στην τοπική βάση, ένα
       ζευγάρι κλειδιά Ed25519 (ο αλγόριθμος αυτός χρησιμοποιείται από το
       BigchainDB)
    const sender = new driver.Ed25519Keypair(
            fromHexadecimalStringtoUint8Array(userS.address_publicKey)
    );

       //δημιουργώ το genesis block στην τοπική μου βάση που αντιστοιχεί σε
       έγγραφο της συλλογής Blockchain
    const genesis = new Blockchain({
        message: 'This is the genesis block',
        senderId: '',
        recipientId: '',
        timestamp: GetDateTimeNow(),
        previousBlockHash: '',
        currentBlockHash:
       'd41cbddc5fe1aeb84e967773845f21a63be45bb5a4e758e57bb453cd4b4768a811cdaaf8
             01ecc5cfb0d826bc20bf14f47d064156c2ce632dc8044ca64bbf4aba',
        index: 0,
        name: 'BlockchainMessenger_v1'
    })

       //δημιουργώ την συναλλαγή για το BigchainDB σύμφωνα με τις
       προκαθορισμένες εντολές
    const transaction = driver.Transaction.makeCreateTransaction(
        genesis,
        null,
        [ driver.Transaction.makeOutput(
            driver.Transaction.makeEd25519Condition(sender.publicKey))],
```

```
        sender.publicKey
    );

    //υπογράφεται η συναλλαγή με το ιδιωτικό κλειδί του αποστολέα
    const  transactionSigned  =  driver.Transaction.signTransaction(transaction,
                                sender.privateKey);

    //στέλνεται η συναλλαγή στο blockchain του BigchainDB και αποθηκεύεται
    conn.postTransactionCommit(transactionSigned);

    //τελικά αν όλα πάνε καλά με το BigchainDB, αποθηκεύεται η συναλλαγή σαν
κουτί στην τοπική μας βάση
    await genesis.save();
}
//ΤΕΛΟΣ ΣΥΝΑΡΤΗΣΕΩΝ BLOCKCHAIN**********************************************

SocketIO.use(async (socket, next) => {

    try{
        const userToken = socket.handshake.query.token; // αφού το έχουμε
              περάσει έτσι στο conversation.js

         // οπότε τώρα πρέπει να κάνω verify το token και να περιμένω να γίνει
         για να συνεχίσω παρακάτω για αυτό είναι σε try/catch και με
         async/await. Όταν το κάνω sign στο userController χρησιμοποιώ το id του
         user (αυτό δηλαδή που δίνει αυτόματα το mongodb σε κάθε εγγραφη). Άρα,
         από το payload.id έχω το id του χρήστη που έχει εισέλθει αυτή τη στιγμή
         την εφαρμογή
        const payload = await jwt.verify(userToken, process.env.SECRET);

        socket.userId = payload.id;

        next();
    } catch(error) {}
});

SocketIO.on('connect', async (socket) => {
    console.log('connected');
       // το socket.userId είναι αυτός που μπαίνει στο conversation room και
       είναι ο αποστολέας, ενώ userId είναι αυτός που θα παραλαβει το μήνυμα
    socket.on('disconnect', () => {
        console.log('disconnected');
    });

    socket.on('enterConversation', async ({conversationId, userId}) => {
        socket.join(conversationId);
        console.log('entered the conversation ' + conversationId)

        const blockchain = await Blockchain.find(); // επιστρέφει όλο το
              blockchain

        const res = await conn.searchAssets('BlockchainMessenger_v1');

         // αν δεν υπάρχει το blockchain και επίσης δεν υπάρχει και στα αρχεία
         του bigchaindb, τότε σημαίνει πως είναι η πρώτη φορά που γίνεται
         εισαγωγή στον πίνακα αυτόν στην βάση, άρα πρέπει να δημιουργήσω το
         genesis block. Αυτό θα γίνει μόνο αν ισχύουν τα παραπάνω και επίσης το
         μήκος του πίνακα από το bigchaindb είναι 0, δηλαδή δεν υπάρχει τέτοιο
         blockchain με αυτό το όνομα
        if(blockchain.length === 0 && res.length === 0) {
```

```
            await CreateGenesisBlock(socket.userId);
        }
        else{
            const valid = await VerifyTheBlockchain(blockchain);
            if(!valid){
                await ReconstructTheBlockchain();
                    SocketIO.to(conversationId).emit('informTheUserWhenEntering',
                     {});
            }
        }

        const userS = await User.findOne({_id: socket.userId}); // αποστολέας
        const userR = await User.findOne({_id: userId}); // παραλήπτης

        const senderKeyPair = nacl.box.keyPair. fromSecretKey(
              fromHexadecimalStringtoUint8Array(userS.address_publicKey));
        const recipientKeyPair = nacl.box.keyPair.fromSecretKey(
              fromHexadecimalStringtoUint8Array(userR.address_publicKey));

        const oneTimeCode1 = recipientKeyPair.publicKey.slice(8);
        const oneTimeCode2 = senderKeyPair.publicKey.slice(8);


        // βρίσκω τα blocks που περιέχουν τα μηνύματα μεταξύ των 2 χρηστών είτε
       αυτά που χει στείλει ο ένας και τα παρέλαβε ο άλλος είτε το αντίστροφο
          const blocks = await Blockchain.find({$or:[ {senderId: socket.userId,
              recipientId: userId}, {senderId: userId, recipientId:
              socket.userId} ] })
        const userMessages = [];
        if(blocks.length>=1){
            for(var block of blocks){
                let decodedMessage = block.senderId === socket.userId?
                     nacl.box.open(
                            fromHexadecimalStringtoUint8Array(block.message),
                            oneTimeCode1,senderKeyPair.publicKey,
                            recipientKeyPair.secretKey)
                     : nacl.box.open(
                            fromHexadecimalStringtoUint8Array(block.message),
                            oneTimeCode2, recipientKeyPair.publicKey,
                            senderKeyPair.secretKey);
                let readableMessage = nacl.util.encodeUTF8(decodedMessage);

                const message = {
                    message: readableMessage,
                    username: block.senderId === socket.userId ?
                     userS.name :  userR.name,
                    userSender: block.senderId
                }
                userMessages.push(message);
            }

            if(userMessages && userS && userR){
                SocketIO.to(conversationId).emit('getAllMessages', {
                    allMessages: userMessages
                });
            }
        }
    });

    socket.on('leaveConversation', (conversationId) => {
```

```
        socket.leave(conversationId);
        console.log('left the conversation ' + conversationId.conversationId) //
              όταν γίνεται emit το leaveConversation στο frontend επιστρέφει
              conversationId σαν αντικείμενο με ιδιότητα το conversationId και
              όχι απλά σαν string
    })
     socket.on('conversationMessage', async ({conversationId, message, userId})
       => {
        // μονο αν το μήνυμα δεν είναι κενό τότε κάνω emit μια νέα μέθοδο
        newMessage
        const userS = await User.findOne({_id: socket.userId}); // αποστολέας
        const userR = await User.findOne({_id: userId}); // παραλήπτης
        var blockchain = await Blockchain.find(); // επιστρέφει όλο blockchain

        //εδώ θα ελέγχω αν είναι έγκυρο το blockchain
        if(message){
            var valid = await VerifyTheBlockchain(blockchain);
            var validAfterRecons;

            console.log('is chain valid?' + valid);
            if(!valid){
                const reconstructed = await ReconstructTheBlockchain();
                if(reconstructed){
                    validAfterRecons = VerifyTheBlockchain(blockchain);
                }
                    SocketIO.to(conversationId).emit('informTheUserWhenSending',
                  {});
                blockchain = await Blockchain.find();
            }

            if(valid || validAfterRecons){
                console.log('chain is valid');
                    // keypair αποστολέα για την συναλλαγή/transaction στο
                     bigchaindb
                const sender = new driver.Ed25519Keypair(
                            fromHexadecimalStringtoUint8Array(
                                   userS.address_publicKey));

                     // τα keypairs που θα χρησιμοποιηθούν για την κρυπτογράφηση
                    σύμφωνα με το δημόσιο κλειδί που έχει δοθεί στον κάθε
                    χρηστη. αυτά δεν αποθηκεύονται για λόγους ασφαλείας, αλλα
                    κάθε φορα γίνονται generate με βάση το δημόσιο κλειδί που
                    υπάρχει στη βάση
                const senderKeyPair = nacl.box.keyPair.fromSecretKey(
              fromHexadecimalStringtoUint8Array(userS.address_publicKey));
                const recipientKeyPair = nacl.box.keyPair.fromSecretKey(
              fromHexadecimalStringtoUint8Array(userR.address_publicKey));

                const oneTimeCode = recipientKeyPair.publicKey.slice(8);

                //ποιο είναι το μήνυμα που γράφει ο χρήστης
                const unencryptedMessage = message;
                //κρυπτογράφησε το
                const encryptedMessage = nacl.box(
                    nacl.util.decodeUTF8(unencryptedMessage),
                    oneTimeCode,
                    recipientKeyPair.publicKey,
                    senderKeyPair.secretKey
                );
```

```
                const hexEncryptedMessage =
                     fromUint8ArraytoHexadecimalString(encryptedMessage);

                const Timestamp = GetDateTimeNow();
                const PreviousBlockHash = blockchain[blockchain.length -    1].
                                                    currentBlockHash;
                const BlockIndex = blockchain[blockchain.length -1].index + 1;
                const CurrentBlockHash = ComputeTheHashOfThisBlock(
                                  hexEncryptedMessage, PreviousBlockHash,
                                  Timestamp, userS._id, userR._id, BlockIndex);

                const assetData_BlockData = {
                    message: hexEncryptedMessage,
                    senderId: userS._id ,
                    recipientId: userR._id,
                    timestamp: Timestamp,
                    previousBlockHash: PreviousBlockHash,
                    currentBlockHash: CurrentBlockHash,
                    index: BlockIndex,
                    name: 'BlockchainMessenger_v1'
                }

                const transaction = driver.Transaction.makeCreateTransaction(
                    assetData_BlockData,
                    null,
                    [ driver.Transaction.makeOutput(
                            driver.Transaction.makeEd25519Condition(
                            sender.publicKey))
                     ],
                    sender.publicKey
                );

                const transactionSigned = driver.Transaction.signTransaction(
                                                transaction, sender.privateKey);

                conn.postTransactionCommit(transactionSigned);

                const newBlock = new Blockchain(assetData_BlockData);

                let decodedMessage = nacl.box.open(
                     fromHexadecimalStringtoUint8Array(hexEncryptedMessage),
                     oneTimeCode, senderKeyPair.publicKey,
                     recipientKeyPair.secretKey);

                let readableMessage = nacl.util.encodeUTF8(decodedMessage);

                  // είναι από τη μεριά αυτού που στέλνει, άρα userId και name
                     αυτού που το στέλνει
                SocketIO.to(conversationId).emit('newConvMessage', {
                    userId: userS._id,
                    name: userS.name,
                    message: readableMessage
                });
                await newBlock.save();
            }
        }
    })
});
```