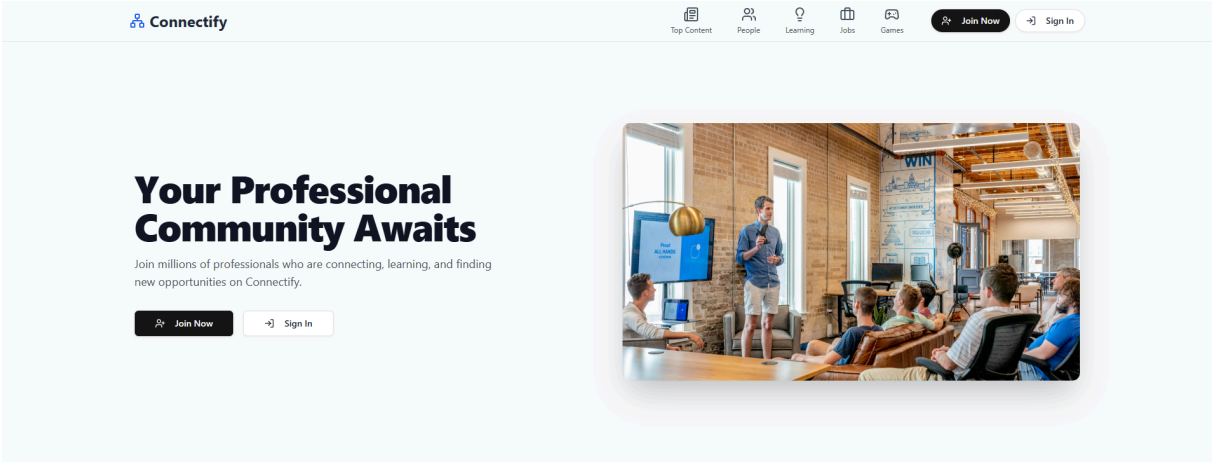
Connectify - Πλατφόρμα Επαγγελματικής Δικτύωσης

Μάθημα: Τεχνολογίες Εφαρμογών Διαδικτύου



Τμήμα Πληροφορικής και Τηλεπικοινωνιών
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Πίνακας Περιεχομένων

1. Εισαγωγή

Η παρούσα εργασία παρουσιάζει τον σχεδιασμό και την υλοποίηση της διαδικτυακής εφαρμογής **Connectify**. Το **Connectify** αποτελεί μια ολοκληρωμένη πλατφόρμα επαγγελματικής δικτύωσης, εμπνευσμένη από καθιερωμένες πλατφόρμες όπως το LinkedIn. Κύριος στόχος του έργου είναι η δημιουργία ενός δυναμικού και διαδραστικού περιβάλλοντος όπου οι επαγγελματίες μπορούν να δημιουργήσουν και να διατηρήσουν το επαγγελματικό τους προφίλ, να συνδεθούν με άλλους χρήστες, να αναζητήσουν ευκαιρίες απασχόλησης, και να μοιραστούν περιεχόμενο σχετικό με τον κλάδο τους. Η πλατφόρμα αναπτύχθηκε με γνώμονα τη σύγχρονη τεχνολογία, την επεκτασιμότητα και την παροχή μιας άρτιας εμπειρίας χρήστη.

Σκοπός της παρούσας τεχνικής αναφοράς είναι η λεπτομερής καταγραφή των σχεδιαστικών αποφάσεων, των αρχιτεκτονικών επιλογών και των τεχνικών υλοποίησης που ακολουθήθηκαν κατά την ανάπτυξη της εφαρμογής. Στις σελίδες που ακολουθούν, αναλύονται οι βασικές παραδοχές που τέθηκαν, οι τεχνολογίες που επιλέχθηκαν για το frontend και το backend, καθώς και οι λόγοι πίσω από τις επιλογές αυτές. Επιπλέον, το έγγραφο λειτουργεί ως οδηγός, παρέχοντας πληροφορίες για την εγκατάσταση, τη ρύθμιση και την εκτέλεση της πλατφόρμας σε ένα τοπικό περιβάλλον ανάπτυξης.

Η δομή της αναφοράς οργανώνεται στα ακόλουθα κεφάλαια:

- **Κεφάλαιο 2 — Αρχιτεκτονική Συστήματος & Στρατηγική Deployment:** Παρουσιάζεται η συνολική δομή της εφαρμογής και η αλληλεπίδραση client-server. Αναλύονται επίσης οι DevOps πρακτικές που υιοθετήθηκαν, όπως containerization με Docker και η συνεχής ολοκλήρωση (CI) με GitHub Actions.
- **Κεφάλαιο 3 — Backend:** Σε αυτό το κεφάλαιο αναλύεται η αρχιτεκτονική της server-side λογικής. Εξετάζονται οι τεχνολογίες που χρησιμοποιήθηκαν, με έμφαση στη γλώσσα προγραμματισμού Elixir και το web framework Phoenix. Περιγράφεται επίσης ο σχεδιασμός της βάσης δεδομένων, η υλοποίηση του RESTful API και ο μηχανισμός αυθεντικοποίησης.
- **Κεφάλαιο 4 — Frontend:** Το κεφάλαιο αυτό εστιάζει στην πλευρά του client. Παρουσιάζεται η δομή της εφαρμογής που αναπτύχθηκε με το framework React και τη γλώσσα TypeScript. Αναλύονται οι βασικές βιβλιοθήκες, η στρατηγική διαχείρισης κατάστασης (state management) και η αρχιτεκτονική των components.
- **Κεφάλαιο 5 — Testing:** Το τελευταίο κεφάλαιο είναι αφιερωμένο στη διασφάλιση της ποιότητας της εφαρμογής. Περιγράφεται η στρατηγική ελέγχου, η υποδομή που έχει στηθεί και τα είδη των tests που υλοποιήθηκαν, με έμφαση στο backend.

2. Αρχιτεκτονική Συστήματος & Στρατηγική Deployment

Πριν την εν τω βάθει ανάλυση των επιμέρους τμημάτων (backend και frontend), σε αυτό το κεφάλαιο παρουσιάζεται η συνολική αρχιτεκτονική της πλατφόρμας **Connectify**. Εξετάζεται ο τρόπος με τον οποίο τα τμήματα της εφαρμογής συνδέονται μεταξύ τους, καθώς και οι στρατηγικές που υιοθετήθηκαν για την αυτοματοποίηση, containerization και τη διασφάλιση της ποιότητας του κώδικα, στοιχεία που είναι θεμελιώδη για τη συντηρησιμότητα και την επεκτασιμότητα του έργου.

2.1. Αρχιτεκτονική Υψηλού Επιπέδου

Η εφαρμογή ακολουθεί μια κλασική αρχιτεκτονική **client-server**. Αποτελείται από δύο κύρια, ανεξάρτητα αναπτυσσόμενα μέρη:

- **Frontend Application:** Μια Single Page Application (SPA) γραμμένη σε React, η οποία εκτελείται εξ ολοκλήρου στον browser του χρήστη. Είναι υπεύθυνη για την απόδοση της διεπαφής χρήστη και την αλληλεπίδραση με αυτόν.
- **Backend Application:** Ένας server γραμμένος σε Elixir/Phoenix, ο οποίος εκθέτει ένα RESTful API και διαχειρίζεται την επικοινωνία μέσω WebSockets. Είναι υπεύθυνος για όλη την επιχειρησιακή λογική, την αυθεντικοποίηση και την επικοινωνία με τη βάση δεδομένων.

Η επικοινωνία μεταξύ τους γίνεται μέσω δύο καναλιών:

1. **Ασύγχρονα HTTP Αιτήματα:** Το frontend χρησιμοποιεί το REST API του backend για τις περισσότερες λειτουργίες CRUD (Create, Read, Update, Delete).
2. **WebSockets:** Για λειτουργίες πραγματικού χρόνου (chat, ειδοποιήσεις, κατάσταση παρουσίας), εγκαθίσταται μια μόνιμη, διπλής κατεύθυνσης σύνδεση WebSocket μέσω των Phoenix Channels.

2.2. Containerization με Docker

Για την απλοποίηση της διαδικασίας εγκατάστασης, τη διασφάλιση ενός συνεκτικού περιβάλλοντος ανάπτυξης για όλους τους προγραμματιστές και την προετοιμασία για ένα παραγωγικό περιβάλλον, η εφαρμογή έχει γίνει πλήρως containerized με τη χρήση **Docker** και **Docker Compose**.

Το αρχείο `docker-compose.yml` στη ρίζα του project ορίζει τις τρεις βασικές υπηρεσίες (services) που απαιτούνται για την εκτέλεση της εφαρμογής:

- `db`: Μια υπηρεσία PostgreSQL που λειτουργεί ως η βάση δεδομένων της εφαρμογής. Τα δεδομένα της αποθηκεύονται σε ένα Docker volume για να παραμένουν ακόμα και μετά τον τερματισμό του container.
- `backend`: Η υπηρεσία που εκτελεί τον Phoenix server. Κάνει build την εφαρμογή Elixir μέσα στο container, εκτελεί τα database migrations και την εκθέτει στην πόρτα 4000.
- `frontend`: Η υπηρεσία που εκτελεί τον Vite development server για την React εφαρμογή και την εκθέτει στην πόρτα 5173, κάνοντας proxy τα API requests προς το backend.

Η χρήση του Docker Compose προσφέρει το σημαντικό πλεονέκτημα της εκκίνησης ολόκληρης της στοίβας τεχνολογιών με μία μόνο εντολή (`docker-compose up`), εξαλείφοντας την ανάγκη για χειροκίνητη εγκατάσταση της Elixir, του Node.js ή της PostgreSQL στο τοπικό μηχάνημα.

2.3. Συνεχής Ολοκλήρωση & Παράδοση (CI/CD) με GitHub Actions

Για την αυτοματοποίηση των διαδικασιών ελέγχου ποιότητας, συντήρησης και δημοσίευσης της εφαρμογής, υιοθετήθηκε μια εξελιγμένη στρατηγική CI/CD με τη χρήση των **GitHub Actions**. Αντί για ένα μονολιθικό αρχείο, δημιουργήθηκαν πολλαπλά, εξειδικευμένα workflows που ενεργοποιούνται σε διαφορετικά events, διασφαλίζοντας την αποτελεσματικότητα και την ταχύτητα της διαδικασίας.

- **`static_analysis.yml`**: Ενεργοποιείται σε κάθε `push` και `pull request` στο `main` branch. Είναι υπεύθυνο για τη στατική ανάλυση του κώδικα και για τα δύο μέρη της εφαρμογής. Για το backend, εκτελεί `mix format --check-formatted`, `mix credo` (ανάλυση στυλ) και `mix sobelow` (ανάλυση ασφαλείας). Για το frontend, εκτελεί `npm run lint`, `npx tsc --noEmit` (έλεγχος τύπων TypeScript) και `npm run format:check`.
- **`tests.yml`**: Ενεργοποιείται επίσης σε `push` και `pull request` και είναι αποκλειστικά υπεύθυνο για την εκτέλεση των αυτοματοποιημένων ελέγχων. Ξεκινά μια υπηρεσία PostgreSQL (`services: postgres`) μέσα στο workflow για να εκτελέσει το test suite του backend (`mix test`) σε ένα καθαρό περιβάλλον.
- **`format-code.yml`**: Ένα workflow που ενεργοποιείται σε pull requests και αυτοματοποιεί τη μορφοποίηση του κώδικα. Ελέγχει αν ο κώδικας είναι σωστά μορφοποιημένος και, αν όχι, εκτελεί `mix format` και `npm run format`, και κάνει commit τις αλλαγές απευθείας στο pull request branch.
- **`update-dependencies.yml`**: Ένα προγραμματισμένο workflow (cron job) που εκτελείται καθημερινά. Ενημερώνει τις εξαρτήσεις του project (`npm update` και `mix deps.update --all`) και, αν υπάρχουν αλλαγές, δημιουργεί ένα commit με τα ενημερωμένα lockfiles.
- **`release.yml`**: Το workflow για τη δημιουργία εκδόσεων (releases). Ενεργοποιείται όταν δημιουργείται ένα νέο tag με τη μορφή `v*` (π.χ., `v1.0.0`). Αυτό το workflow κάνει build το frontend για production, ενσωματώνει τα στατικά αρχεία μέσα στην Phoenix εφαρμογή, δημιουργεί ένα production release του backend (`mix release`) και το πακετάρει σε ένα `.tar.gz` αρχείο, το οποίο επισυνάπτει σε ένα νέο GitHub Release.

3. Backend

Το backend της πλατφόρμας **Connectify** αποτελεί τη ραχοκοκαλιά της εφαρμογής, διαχειριζόμενο το σύνολο της επιχειρησιακής λογικής, την επικοινωνία με τη βάση δεδομένων, την αυθεντικοποίηση των χρηστών, καθώς και την παροχή ενός RESTful API για την επικοινωνία με το frontend. Η επιλογή της τεχνολογικής στοίβας έγινε με γνώμονα την απόδοση, την επεκτασιμότητα και την ικανότητα διαχείρισης ταυτόχρονων συνδέσεων σε πραγματικό χρόνο, χαρακτηριστικά απαραίτητα για μια σύγχρονη πλατφόρμα κοινωνικής δικτύωσης.

3.1. Τεχνολογική Στοίβα (Technology Stack)

Η ανάπτυξη του backend βασίστηκε στις εξής κύριες τεχνολογίες:

- **Γλώσσα Προγραμματισμού: Elixir.** Η Elixir επιλέχθηκε λόγω της λειτουργικής της φύσης και της εκτέλεσής της πάνω στην Erlang Virtual Machine (BEAM). Αυτό της προσδίδει εξαιρετική ικανότητα διαχείρισης χιλιάδων ταυτόχρονων διεργασιών (concurrency) με χαμηλό κόστος, καθώς και υψηλή ανθεκτικότητα σε σφάλματα (fault tolerance), καθιστώντας την ιδανική για εφαρμογές πραγματικού χρόνου όπως chat, ειδοποιήσεις και live status χρηστών.
- **Web Framework: Phoenix.** Το Phoenix Framework, χτισμένο πάνω στην Elixir, παρέχει ένα δομημένο και παραγωγικό περιβάλλον για την ανάπτυξη web εφαρμογών. Αξιοποιήθηκαν τα βασικά του χαρακτηριστικά, όπως:
  - **Phoenix Channels:** Για την υλοποίηση της real-time επικοινωνίας μέσω WebSockets, που είναι κρίσιμη για το chat, τις ειδοποιήσεις και την κατάσταση παρουσίας των χρηστών.
  - **Phoenix Contexts:** Για την οργάνωση της επιχειρησιακής λογικής σε διακριτά, απομονωμένα modules (`Accounts`, `Posts`, `Jobs` κ.ά.), προωθώντας τη συντηρησιμότητα και την καθαρότητα του κώδικα.
  - **Ecto:** Ως βιβλιοθήκη για την επικοινωνία με τη βάση δεδομένων, προσφέροντας ένα ισχυρό DSL για την εκτέλεση ερωτημάτων και `changesets` για την επικύρωση και τον μετασχηματισμό των δεδομένων.
- **Βάση Δεδομένων: PostgreSQL.** Επιλέχθηκε ως το σύστημα διαχείρισης σχεσιακής βάσης δεδομένων λόγω της σταθερότητας, της απόδοσης και της υποστήριξης προηγμένων τύπων δεδομένων και δεικτών, που είναι απαραίτητα για τις πολύπλοκες σχέσεις μεταξύ των οντοτήτων της εφαρμογής.
- **Αυθεντικοποίηση: JWT (JSON Web Tokens).** Η αυθεντικοποίηση των χρηστών υλοποιήθηκε μέσω JWT, με τη χρήση της βιβλιοθήκης `Joken`. Μετά την επιτυχή είσοδο, ο server υπογράφει ένα token το οποίο αποθηκεύεται τοπικά στον client και αποστέλλεται σε κάθε επόμενο αίτημα για την ταυτοποίηση του χρήστη. Η κρυπτογράφηση των κωδικών πρόσβασης γίνεται με τον αλγόριθμο `Argon2`, μέσω της βιβλιοθήκης `argon2_elixir`, για μέγιστη ασφάλεια.
- **Web Server: Bandit.** Ως web server για το Phoenix χρησιμοποιήθηκε ο `Bandit`, μια σύγχρονη και αποδοτική υλοποίηση γραμμένη αμιγώς σε Elixir, που προσφέρει βελτιωμένη απόδοση σε σχέση με τον προεπιλεγμένο server Cowboy.

3.2. Αρχιτεκτονική

Η αρχιτεκτονική του backend ακολουθεί το **Context Pattern** που προωθεί το Phoenix Framework. Η λογική της εφαρμογής είναι διαχωρισμένη σε αυτόνομα modules (contexts), καθένα από τα οποία είναι υπεύθυνο για ένα συγκεκριμένο τομέα της εφαρμογής.

- **Context Modules:** Κάθε context (π.χ., `Backend.Accounts`, `Backend.Posts`, `Backend.Jobs`, `Backend.Connections`) ομαδοποιεί τη σχετική επιχειρησιακή λογική και τα σχήματα της βάσης δεδομένων. Για παράδειγμα, το `Backend.Accounts` διαχειρίζεται τα πάντα που αφορούν τους

χρήστες: δημιουργία, αυθεντικοποίηση, ενημέρωση προφίλ κ.λπ. Αυτή η προσέγγιση καθιστά την εφαρμογή πιο αρθρωτή και εύκολη στη συντήρηση.

- **RESTful API:** Το backend εκθέτει ένα RESTful API για την επικοινωνία με το frontend. Οι `Controllers` (π.χ., `PostController`, `UserController`) είναι υπεύθυνοι για τη λήψη των HTTP αιτημάτων, την κλήση των κατάλληλων συναρτήσεων από τα context modules και τη διαμόρφωση της απάντησης. Για τη σειριοποίηση των δεδομένων σε μορφή JSON χρησιμοποιούνται τα `JSON Views` (π.χ., `PostJSON`, `UserJSON`), τα οποία ορίζουν τη δομή των απαντήσεων.
- **Real-time Layer (Phoenix Channels):** Παράλληλα με το REST API, υπάρχει ένα επίπεδο επικοινωνίας πραγματικού χρόνου που υλοποιείται με Phoenix Channels. Ο `UserSocket` (`lib/backend_web/channels/user_socket.ex`) διαχειρίζεται την αρχική σύνδεση του client μέσω WebSocket. Στη συνέχεια, ο client μπορεί να κάνει `join` σε συγκεκριμένα κανάλια (topics):
  - `ChatChannel`: Για την αποστολή και λήψη μηνυμάτων σε πραγματικό χρόνο.
  - `StatusChannel`: Για τη διαχείριση της κατάστασης παρουσίας των χρηστών (online, idle, offline).
  - `NotificationsChannel`: Για την άμεση προώθηση νέων ειδοποιήσεων στους χρήστες.

3.3. Σχεδιασμός Βάσης Δεδομένων

Ο σχεδιασμός της βάσης δεδομένων αποτελεί θεμελιώδες κομμάτι της εφαρμογής. Ως πρωτεύοντα κλειδιά για όλους τους πίνακες χρησιμοποιήθηκε ο τύπος `binary_id` (UUID), μια συνήθης πρακτική σε σύγχρονες εφαρμογές για την αποφυγή συγκρούσεων σε κατανεμημένα συστήματα. Οι κυριότεροι πίνακες και οι σχέσεις τους είναι:

- `users`: Ο κεντρικός πίνακας που αποθηκεύει τα στοιχεία των χρηστών. Περιέχει πεδία για αυθεντικοποίηση (`email`, `password_hash`), προσωπικές πληροφορίες (`name`, `surname`, `photo_url`) και ρυθμίσεις (`role`, `profile_visibility`).
- `companies`: Αποθηκεύει τις εταιρείες της πλατφόρμας.
- `job_experiences` & `educations`: Συνδέονται με τους χρήστες (many-to-one) και περιγράφουν το επαγγελματικό και ακαδημαϊκό τους υπόβαθρο.
- `skills`: Πίνακας με τις δεξιότητες. Υπάρχει μια σχέση many-to-many με τους χρήστες (`users_skills`) και τα job postings (`job_postings_skills`).
- `connections`: Υλοποιεί τις συνδέσεις μεταξύ των χρηστών (self-referencing many-to-many). Ένα πεδίο `status` διακρίνει τις εκκρεμείς από τις αποδεκτές συνδέσεις.
- `posts`, `comments`, `reactions`: Οι βασικές οντότητες του news feed. Ένα post ανήκει σε έναν χρήστη. Ένα comment ανήκει σε ένα post. Μια αντίδραση ανήκει σε ένα post και έναν χρήστη.
- `job_postings` & `job_applications`: Οι αγγελίες εργασίας και οι αιτήσεις των χρηστών σε αυτές.
- `interests`: Υλοποιεί το σύστημα “follow” για χρήστες και εταιρείες, χρησιμοποιώντας ένα πεδίο `type` για να διακρίνει την οντότητα που ακολουθείται.

3.4. Υλοποίηση Βασικών Λειτουργιών

3.4.1. Αυθεντικοποίηση και Εξουσιοδότηση

Η διαδικασία αυθεντικοποίησης ξεκινά από τον `SessionController`, ο οποίος, μετά την επιτυχή επαλήθευση των στοιχείων του χρήστη, καλεί το `Backend.Auth.sign_token/1` για τη δημιουργία ενός JWT. Αυτό το token αποθηκεύεται σε ένα `secure`, `httpOnly` cookie.

Για την εξουσιοδότηση, το `AuthPlug` (`lib/backend_web/plugs/auth_plug.ex`) εκτελείται σε κάθε αίτημα προς το API. Αναλύει το JWT από το cookie, επαληθεύει την υπογραφή του και, αν είναι έγκυρο, ανακτά τον αντίστοιχο χρήστη από τη βάση δεδομένων και τον επισυνάπτει στο `conn` (connection struct), καθιστώντας τον διαθέσιμο σε όλους τους controllers.

Για τις διαδρομές που απαιτούν δικαιώματα διαχειριστή (π.χ., `/api/admin/*`), χρησιμοποιείται ένα επιπλέον plug, το `EnsureAdminPlug`, το οποίο ελέγχει αν ο `current_user` έχει τον ρόλο "admin" και σε αντίθετη περίπτωση διακόπτει το αίτημα με κωδικό σφάλματος 403 Forbidden.

3.4.2. Μηχανισμός Προτάσεων (Recommendation Engine)

Η πλατφόρμα ενσωματώνει ένα σύστημα προτάσεων για αγγελίες εργασίας και αναρτήσεις, το οποίο υλοποιείται στο `Backend.Recommendations.Recommender`. Ο αλγόριθμος που χρησιμοποιείται είναι μια μορφή **Συνεργατικού Φιλτραρίσματος (Collaborative Filtering)**, και συγκεκριμένα η τεχνική **Παραγοντοποίησης Πινάκων (Matrix Factorization)**.

- **Προτάσεις Εργασίας:** Ο αλγόριθμος αναλύει τις αιτήσεις (`JobApplication`) που έχουν κάνει όλοι οι χρήστες σε όλες τις αγγελίες, δημιουργώντας έναν πίνακα "χρήστης-αντικείμενο". Μέσω της παραγοντοποίησης, προβλέπει ποιες αγγελίες θα ενδιέφεραν έναν χρήστη με βάση τις αιτήσεις που έχουν κάνει άλλοι "παρόμοιοι" χρήστες.
- **Προτάσεις Αναρτήσεων:** Η λογική είναι παρόμοια, αλλά ο πίνακας "χρήστης-αντικείμενο" κατασκευάζεται από τις αλληλεπιδράσεις των χρηστών με τις αναρτήσεις, όπως reactions, comments και views. Κάθε τύπος αλληλεπίδρασης έχει διαφορετικό βάρος (`@reaction_weight`, `@comment_weight`, `@view_weight`) για τον υπολογισμό του "ενδιαφέροντος".

3.4.3. Προηγμένες Λειτουργίες Ροής Αγγελιών και Αναρτήσεων

Πέρα από την απλή εμφάνιση, η ροή περιεχομένου (τόσο οι αγγελίες εργασίας όσο και οι αναρτήσεις) εμπλουτίστηκε με λογική που αυξάνει τη συνάφεια για τον χρήστη, παρόμοια με αυτήν καθιερωμένων πλατφορμών.

- **Σχετικές Συνδέσεις σε Αγγελίες:** Στη ροή αγγελιών (`Backend.Jobs.list_job_postings_for_user_feed/1`), το σύστημα ελέγχει αν κάποια από τις συνδέσεις του τρέχοντος χρήστη εργάζεται ήδη στην εταιρεία που δημοσίευσε την αγγελία. Αυτή η πληροφορία επιστρέφεται στο frontend, παρέχοντας στον χρήστη ένα πολύτιμο "insider" πλεονέκτημα.
- **Ταξινόμηση Βάσει Συνάφειας:** Τόσο οι αγγελίες όσο και οι αναρτήσεις ταξινομούνται με έναν υβριδικό αλγόριθμο. Για τις αγγελίες, λαμβάνεται υπόψη ο αριθμός των κοινών δεξιοτήτων (`matching skills`) του χρήστη με την αγγελία. Για τις αναρτήσεις, η συνάρτηση `calculate_post_score` (`Backend.Posts`) υπολογίζει ένα σκορ συνάφειας που συνυπολογίζει την παλαιότητα, τη δημοτικότητα (προβολές) και το αν η ανάρτηση προέρχεται από άτομο του δικτύου του χρήστη.

3.4.4. Εμπλουτισμένες Λειτουργίες Πραγματικού Χρόνου

Οι δυνατότητες των Phoenix Channels αξιοποιήθηκαν στο έπακρο για να προσφέρουν μια πλούσια διαδραστική εμπειρία που ξεπερνά την απλή ανταλλαγή μηνυμάτων.

- **Ανταλλαγή Πολυμέσων στο Chat:** Οι χρήστες μπορούν να ανεβάζουν και να στέλνουν εικόνες και αρχεία στις ιδιωτικές τους συζητήσεις. Το backend διαχειρίζεται την αποθήκευση των αρχείων και προωθεί τα σχετικά URLs μέσω των WebSockets.
- **Αντιδράσεις (Reactions) σε Μηνύματα:** Προστέθηκε η δυνατότητα αντίδρασης σε συγκεκριμένα μηνύματα του chat, επιτρέποντας μια πιο εκφραστική επικοινωνία.
- **Κοινοποίηση Αναρτήσεων ως Μήνυμα:** Οι χρήστες μπορούν να μοιραστούν μια δημόσια ανάρτηση από τη ροή ειδήσεων απευθείας σε μια ιδιωτική συζήτηση, μια λειτουργία που υλοποιείται μέσω του `Chat.send_post_as_message/3`.

3.4.5. Διαχείριση Ορατότητας Προφίλ και Σύστημα "Follow"

Για την παροχή μεγαλύτερου ελέγχου στους χρήστες, υλοποιήθηκαν δύο διακριτά συστήματα δικτύωσης.

- **Ορατότητα Προφίλ:** Οι χρήστες μπορούν να επιλέξουν αν το προφίλ τους θα είναι "public" ή "connections_only". Η λογική αυτή εφαρμόζεται στο `Accounts.get_user_profile!/2`, το οποίο φιλτράρει τα πεδία που επιστρέφονται στο frontend ανάλογα με τη ρύθμιση του χρήστη και το αν ο θεατής είναι ήδη συνδεδεμένος μαζί του.
- **Σύστημα Follow:** Πέρα από τις αμφίδρομες συνδέσεις, οι χρήστες μπορούν να "ακολουθούν" (follow) άλλους χρήστες ή εταιρείες για να βλέπουν τις ενημερώσεις τους, χωρίς να απαιτείται αμοιβαία αποδοχή. Αυτή η λειτουργία υλοποιήθηκε μέσω του context `Interests`.

3.5. Εγκατάσταση και Εκτέλεση

Υπάρχουν δύο τρόποι για την εγκατάσταση και εκτέλεση του backend σε τοπικό περιβάλλον.

3.5.1. Προτεινόμενη Μέθοδος με Docker

Αυτή είναι η απλούστερη και προτεινόμενη μέθοδος, καθώς δεν απαιτεί την τοπική εγκατάσταση της Elixir ή της PostgreSQL.

1. **Προαπαιτούμενο:** Εγκατάσταση του **Docker** και του **Docker Compose**.
2. **Εκκίνηση:** Από τον ριζικό φάκελο του project, εκτελέστε την εντολή: `docker-compose up --build` Αυτή η εντολή θα δημιουργήσει τα images, θα ξεκινήσει όλες τις υπηρεσίες και θα αρχικοποιήσει τη βάση δεδομένων. Ο server θα είναι διαθέσιμος στη διεύθυνση `https://localhost:4000`.

3.5.2. Χειροκίνητη Μέθοδος

1. **Προαπαιτούμενα:**
   - Εγκατάσταση της **Elixir** (έκδοση > 1.17) και **Erlang OTP** (έκδοση 27).
   - Εγκατάσταση και εκτέλεση του **PostgreSQL**.
2. **Ρύθμιση:**
   - Πλοήγηση στον φάκελο `backend`.
   - Δημιουργία αρχείου `.env` και ρύθμιση του `DATABASE_URL`.
   - Εκτέλεση της εντολής `mix deps.get`.
3. **Βάση Δεδομένων:** Εκτέλεση της εντολής `mix ecto.setup`.
4. **Εκκίνηση Server:** Εκτέλεση της εντολής `mix phx.server`.

4. Frontend

Το frontend της πλατφόρμας **Connectify** υλοποιήθηκε ως μια σύγχρονη Single Page Application (SPA), με κύριο στόχο την παροχή μιας γρήγορης, διαδραστικής και αποκριτικής εμπειρίας χρήστη. Η αρχιτεκτονική του σχεδιάστηκε για να είναι ευέλικτη, συντηρήσιμη και ικανή να διαχειρίζεται αποτελεσματικά την κατάσταση της εφαρμογής, τόσο την τοπική όσο και την καθολική (global state), ενώ παράλληλα επικοινωνεί αδιάλειπτα με το backend API για την ανάκτηση και αποστολή δεδομένων.

4.1. Τεχνολογική Στοίβα (Technology Stack)

Η επιλογή των τεχνολογιών για το frontend έγινε με βάση τις σύγχρονες βέλτιστες πρακτικές και το πλούσιο οικοσύστημα εργαλείων που προσφέρουν.

- **Framework & Γλώσσα: React & TypeScript.** Η React επιλέχθηκε ως η βασική βιβλιοθήκη για την κατασκευή του user interface, λόγω της component-based αρχιτεκτονικής της, του τεράστιου οικοσυστήματος και της μεγάλης κοινότητας υποστήριξης. Η χρήση της TypeScript προσφέρει στατική τυποποίηση (static typing), βελτιώνοντας δραστικά την ποιότητα του κώδικα, μειώνοντας τα σφάλματα κατά το runtime και διευκολύνοντας τη συντήρηση και την αναδιαμόρφωση (refactoring) της εφαρμογής.
- **Build Tool: Vite.** Για τη διαδικασία του development και του bundling, επιλέχθηκε το Vite. Το Vite προσφέρει ταχύτατους χρόνους εκκίνησης του development server και σχεδόν άμεσο Hot Module Replacement (HMR) αξιοποιώντας τα native ES Modules του browser, προσφέροντας μια σημαντικά βελτιωμένη εμπειρία ανάπτυξης σε σχέση με παραδοσιακά εργαλεία όπως το Webpack.
- **Styling: Tailwind CSS & shadcn/ui.** Για τη διαμόρφωση της εμφάνισης της εφαρμογής, χρησιμοποιήθηκε το utility-first CSS framework Tailwind CSS. Αυτή η προσέγγιση επιτρέπει τη γρήγορη δημιουργία σύνθετων και αποκριτικών διεπαφών απευθείας μέσα στο JSX markup. Η βιβλιοθήκη componentes **shadcn/ui** επιλέχθηκε για την παροχή έτοιμων, προσβάσιμων και πλήρως παραμετροποιήσιμων UI components (όπως `Card`, `Dialog`, `Button`), επιταχύνοντας σημαντικά την ανάπτυξη. Η συνάρτηση `cn` (`lib/utils.ts`) χρησιμοποιείται για τη δυναμική συγχώνευση κλάσεων του Tailwind.
- **Client-Server Επικοινωνία:**
  - **Axios:** Για την πραγματοποίηση HTTP αιτημάτων προς το backend RESTful API, χρησιμοποιήθηκε η βιβλιοθήκη Axios. Έχει διαμορφωθεί ένα κεντρικό service layer (`services/*.ts`) που αφαιρεί τη λογική των API calls από τα components.
  - **Phoenix JS Client:** Για την επικοινωνία σε πραγματικό χρόνο, χρησιμοποιήθηκε η επίσημη JavaScript βιβλιοθήκη του Phoenix για τη σύνδεση με τα Phoenix Channels του backend, υποστηρίζοντας λειτουργίες όπως το chat και οι ζωντανές ειδοποιήσεις.
- **Διαχείριση Φορμών: React Hook Form & Zod.** Για τη διαχείριση των φορμών και την επικύρωση των δεδομένων τους, επιλέχθηκε ο συνδυασμός των `react-hook-form` για τη διαχείριση της κατάστασης της φόρμας και `Zod` για τον ορισμό και την επιβολή σχημάτων επικύρωσης.

4.2. Αρχιτεκτονική

Η αρχιτεκτονική του frontend είναι δομημένη γύρω από τις αρχές της αρθρωτότητας και του διαχωρισμού αρμοδιοτήτων (separation of concerns).

- **Δομή Φακέλων:** Ο κώδικας είναι οργανωμένος σε λογικούς φακέλους:

- `pages`: Περιέχει τα top-level components για κάθε σελίδα της εφαρμογής (π.χ., `Homepage.tsx`, `Profile.tsx`).
- `components`: Περιέχει επαναχρησιμοποιήσιμα UI components, ομαδοποιημένα ανά λειτουργικότητα (π.χ., `posts`, `chat`, `common`).
- `contexts`: Υλοποιεί τη διαχείριση της καθολικής κατάστασης μέσω του React Context API.
- `services`: Απομονώνει τη λογική επικοινωνίας με το backend API.
- `hooks`: Περιέχει custom React hooks (π.χ., `useDebounce`).
- `types`: Ορίζει τις TypeScript data structures που χρησιμοποιούνται σε όλη την εφαρμογή.

- **Routing:** Η πλοήγηση στην εφαρμογή διαχειρίζεται από τη βιβλιοθήκη `react-router-dom`. Το αρχείο `routes/routes.tsx` ορίζει όλες τις διαδρομές, αντιστοιχίζοντας κάθε URL path σε ένα συγκεκριμένο page component. Υποστηρίζονται και προστατευμένες διαδρομές (protected routes) μέσω του `ProtectedRoute` component, το οποίο ελέγχει αν ο χρήστης είναι αυθεντικοποιημένος και αν έχει τον απαιτούμενο ρόλο (`professional` ή `admin`) για να αποκτήσει πρόσβαση.

- **Διαχείριση Κατάστασης (State Management):** Η εφαρμογή υιοθετεί μια διπλή στρατηγική για τη διαχείριση της κατάστασης:
  - **Τοπική Κατάσταση (Local State):** Για την κατάσταση που αφορά μεμονωμένα components, χρησιμοποιούνται τα standard React hooks `useState` και `useEffect`.
  - **Καθολική Κατάσταση (Global State):** Για δεδομένα που πρέπει να είναι προσβάσιμα από πολλά components σε διαφορετικά σημεία της εφαρμογής, αξιοποιείται το **React Context API**. Έχουν δημιουργηθεί τα εξής contexts:
    - `AuthContext`: Διαχειρίζεται την κατάσταση αυθεντικοποίησης, τα δεδομένα του συνδεδεμένου χρήστη και το JWT.
    - `PresenceContext`: Συνδέεται στο `StatusChannel` του backend για να παρακολουθεί και να παρέχει την κατάσταση παρουσίας (online/idle/offline) των χρηστών σε πραγματικό χρόνο.
    - `NotificationsContext`: Διαχειρίζεται τη λήψη και την κατάσταση (διαβασμένες/αδιάβαστες) των ειδοποιήσεων.

4.3. Υλοποίηση Βασικών Λειτουργιών

4.3.1. Real-time Λειτουργίες (Chat, Presence, Notifications)

Η διαδραστικότητα της πλατφόρμας ενισχύεται από τις real-time λειτουργίες που υλοποιούνται μέσω WebSockets και του Phoenix JS client.

- **Chat:** Το `ChatWindow.tsx` component, αφού λάβει το `chatRoomId`, συνδέεται σε ένα δυναμικό topic του `ChatChannel` (π.χ., `chat:ROOM_ID`). "Ακούει" για το event `new_msg` για να εμφανίσει νέα μηνύματα και στέλνει (pushes) τα δικά του μηνύματα και typing events στον server.
- **Presence:** Το `PresenceContext` συνδέεται στο `StatusChannel` και χρησιμοποιεί το ενσωματωμένο `Phoenix.Presence` utility για να συγχρονίσει την κατάσταση όλων των συνδεδεμένων χρηστών. Παρέχει μια συνάρτηση `getUserStatus(userId)` που μπορεί να χρησιμοποιηθεί οπουδήποτε στην εφαρμογή για την εμφάνιση της κατάστασης ενός χρήστη.
- **Notifications:** Αντίστοιχα, το `NotificationsContext` συνδέεται στο `NotificationsChannel` και λαμβάνει άμεσα τις νέες ειδοποιήσεις, ενημερώνοντας το UI (π.χ., το καμπανάκι ειδοποιήσεων) χωρίς να απαιτείται ανανέωση της σελίδας.

4.3.2. Διαδικασία Onboarding Νέων Χρηστών

Για τη βελτίωση της εμπειρίας των νέων χρηστών και την ενθάρρυνση της συμπλήρωσης του προφίλ τους, δημιουργήθηκε μια καθοδηγούμενη, πολυ-βηματική διαδικασία onboarding (`pages/auth/Onboarding.tsx`). Αμέσως μετά την εγγραφή, ο χρήστης οδηγείται σε μια σειρά από οθόνες όπου καλείται:

1. Να δηλώσει την τοποθεσία του.
2. Να προσθέσει την πιο πρόσφατη επαγγελματική του εμπειρία.
3. Να επιβεβαιώσει το email του μέσω ενός κωδικού.
4. Να ανεβάσει μια φωτογραφία προφίλ.
5. Να ακολουθήσει προτεινόμενες εταιρείες για να εμπλουτίσει άμεσα τη ροή ειδήσεών του.

Αυτή η διαδικασία διασφαλίζει ότι οι χρήστες έχουν ένα πιο πλήρες προφίλ από την πρώτη στιγμή, βελτιώνοντας τη συνολική ποιότητα των δεδομένων στην πλατφόρμα.

4.3.3. Ολοκληρωμένο Πάνελ Διαχείρισης (Admin Panel)

Για τον ρόλο του διαχειριστή, υλοποιήθηκε ένα πλήρες και λειτουργικό περιβάλλον διαχείρισης, προσβάσιμο από τις διαδρομές `/admin/*`. Το πάνελ αυτό, που προστατεύεται από το `ProtectedRoute` και το `AdminLayout`, παρέχει κεντρική εποπτεία και έλεγχο πάνω σε όλες τις βασικές οντότητες της εφαρμογής. Οι κυριότερες λειτουργίες του περιλαμβάνουν:

- **Dashboard:** Μια αρχική οθόνη με βασικά στατιστικά στοιχεία (συνολικοί χρήστες, συνδέσεις, αγγελίες, αναρτήσεις).
- **Διαχείριση Χρηστών:** Πίνακας με όλους τους χρήστες, με δυνατότητα αλλαγής του ρόλου τους (professional/admin) και προβολής αναλυτικών στοιχείων για τον καθένα.
- **Διαχείριση Εταιρειών, Δεξιοτήτων & Αγγελιών:** Πλήρεις λειτουργίες CRUD (Create, Read, Update, Delete) για τις βασικές αυτές οντότητες.
- **Εξαγωγή Δεδομένων:** Μια εξειδικευμένη σελίδα (`ExportPage.tsx`) που επιτρέπει στον διαχειριστή να επιλέξει συγκεκριμένους (ή όλους τους) χρήστες και να εξάγει τα δεδομένα τους σε μορφή JSON ή XML, καλύπτοντας πλήρως την απαίτηση 4 της εκφώνησης.

4.4. Εγκατάσταση και Εκτέλεση

Όπως και με το backend, υπάρχουν δύο τρόποι για την εκτέλεση του frontend.

4.4.1. Προτεινόμενη Μέθοδος με Docker

Η εκτέλεση της εντολής `docker-compose up` από τον ριζικό φάκελο του project (όπως περιγράφεται στην ενότητα του backend) ξεκινά αυτόματα και τον Vite development server. Θα είναι διαθέσιμος στη διεύθυνση `https://localhost:5173`.

4.4.2. Χειροκίνητη Μέθοδος

1. **Προαπαιτούμενα:** Εγκατάσταση του **Node.js** (έκδοση 20) και **npm**.
2. **Ρύθμιση:**
   - Πλοήγηση στον φάκελο `frontend`.
   - Δημιουργία αρχείου `.env` με το `VITE_GIPHY_API_KEY`.
   - Εκτέλεση της εντολής `npm install`.
3. **Εκκίνηση Server:**
   - Εκτέλεση της εντολής `npm run dev`.

5. Testing

Η διασφάλιση της ποιότητας, της ορθότητας και της σταθερότητας της εφαρμογής **Connectify** αποτέλεσε κεντρικό πυλώνα της διαδικασίας ανάπτυξης. Για τον σκοπό αυτό, υιοθετήθηκε μια στρατηγική ελέγχου που περιλαμβάνει πολλαπλά επίπεδα, με έμφαση στους αυτοματοποιημένους ελέγχους (automated tests) τόσο για το backend όσο και για το frontend. Η προσέγγιση αυτή διασφαλίζει ότι νέες λειτουργίες δεν "σπάνε" τις υπάρχουσες (regression testing) και ότι κάθε κομμάτι της εφαρμογής λειτουργεί όπως αναμένεται.

5.1. Backend Testing

Για τον έλεγχο του backend, το Phoenix Framework παρέχει ένα ολοκληρωμένο testing suite που βασίζεται στο ενσωματωμένο testing framework της Elixir, το **ExUnit**. Οι έλεγχοι οργανώνονται παράλληλα με τη δομή του κώδικα της εφαρμογής μέσα στον φάκελο `test/`. Έχει επιτευχθεί **100% test coverage** για όλη την επιχειρησιακή λογική και τους controllers.

5.1.1. Τύποι Ελέγχων

- **Unit Tests:** Οι έλεγχοι αυτοί εστιάζουν σε μεμονωμένες, μικρές μονάδες κώδικα (συναρτήσεις και modules) με απομονωμένο τρόπο. Για παράδειγμα, ελέγχθηκε η ορθότητα των συναρτήσεων επικύρωσης στα `Ecto.Changeset` των σχημάτων (π.χ., `User.changeset`), διασφαλίζοντας ότι οι κανόνες για τα δεδομένα (π.χ., ελάχιστο μήκος κωδικού) εφαρμόζονται σωστά.
- **Integration Tests (Context Tests):** Το μεγαλύτερο μέρος των ελέγχων αφορά τα context modules (`test/backend/accounts_test.exs`, `test/backend/posts_test.exs` κ.λπ.). Αυτοί οι έλεγχοι δεν απομονώνουν τη βάση δεδομένων, αλλά αλληλεπιδρούν με αυτήν για να επιβεβαιώσουν την ορθή λειτουργία της επιχειρησιακής λογικής στο σύνολό της. Για παράδειγμα, ένα test case για τη δημιουργία ενός post (`Posts.create_post/2`) θα εισάγει όντως μια εγγραφή στη βάση δεδομένων και θα επιβεβαιώσει ότι τα δεδομένα που επιστρέφονται είναι σωστά. Το Ecto Sandbox διασφαλίζει ότι κάθε test case εκτελείται μέσα σε μια απομονωμένη transaction, η οποία γίνεται rollback στο τέλος, αφήνοντας τη βάση δεδομένων σε καθαρή κατάσταση για το επόμενο test.
- **Controller Tests:** Οι έλεγχοι αυτοί προσομοιώνουν HTTP αιτήματα προς το API και εξετάζουν την απάντηση. Ελέγχουν αν ο controller επιστρέφει το σωστό status code (π.χ., `200 OK`, `401 Unauthorized`), τα σωστά headers και ένα JSON σώμα με την αναμενόμενη δομή. Αυτό διασφαλίζει ότι τα endpoints του API λειτουργούν σωστά και είναι ασφαλή. Για παράδειγμα, ελέγχεται αν ένα αίτημα για διαγραφή του post ενός άλλου χρήστη αποτυγχάνει με status 403 Forbidden.
- **Channel Tests:** Υλοποιήθηκαν έλεγχοι και για τα Phoenix Channels (`test/backend_web/channels/`). Αυτά τα tests επιβεβαιώνουν τη λογική εξουσιοδότησης (ποιος μπορεί να συνδεθεί σε ένα κανάλι) και την ορθή διαχείριση των μηνυμάτων που αποστέλλονται και λαμβάνονται μέσω WebSockets.

5.1.2. Εργαλεία και Ρύθμιση

- **ExUnit:** Το βασικό testing framework της Elixir.
- **Ecto Sandbox:** Επιτρέπει την εκτέλεση των tests παράλληλα και με ασφάλεια, απομονώνοντας τις αλλαγές στη βάση δεδομένων ανά test. Η ρύθμιση γίνεται στο `config/test.exs`.
- **Mix Aliases:** Στο αρχείο `mix.exs`, έχει οριστεί ένα alias `test` (`"ecto.create --quiet", "ecto.migrate --quiet", "test"`) που αυτοματοποιεί τη διαδικασία προετοιμασίας της test database πριν την εκτέλεση των ελέγχων.

5.2. Frontend Testing

Για το frontend, έχει οριστεί μια στρατηγική ελέγχου που εστιάζει στην οπτική ορθότητα, τη λειτουργικότητα των components και την ορθή αλληλεπίδραση του χρήστη με το UI. Η υποδομή για την εκτέλεση των ελέγχων έχει διαμορφωθεί πλήρως, και η υλοποίηση των επιμέρους test cases αποτελεί το επόμενο στάδιο της ανάπτυξης, όπως φαίνεται από τα σχολιασμένα βήματα στο workflow `tests.yml`.

5.2.1. Τύποι Ελέγχων που θα Υλοποιηθούν

- **Component Tests:** Η στρατηγική περιλαμβάνει τον έλεγχο μεμονωμένων React components. Χρησιμοποιώντας το **React Testing Library**, τα components θα αποδίδονται (render) σε ένα προσομοιωμένο DOM περιβάλλον. Στη συνέχεια, θα γίνονται assertions για το περιεχόμενο που εμφανίζεται. Για παράδειγμα, ένας έλεγχος για το `Button` component θα επιβεβαιώσει ότι το κείμενο που του δίνεται ως prop εμφανίζεται σωστά και ότι το `onClick` event ενεργοποιείται όταν προσομοιώνεται ένα κλικ.
- **Integration Tests:** Προβλέπεται η υλοποίηση ελέγχων ολοκλήρωσης που θα συνδυάζουν πολλαπλά components για την εξέταση πιο σύνθετων ροών εργασίας. Για παράδειγμα, θα ελεγχθεί η λειτουργία της φόρμας εγγραφής (`Register.tsx`), προσομοιώνοντας την πληκτρολόγηση του χρήστη στα πεδία εισόδου, το κλικ στο κουμπί υποβολής και την επαλήθευση ότι αποστέλλεται το σωστό αίτημα στο backend.

5.2.2. Εργαλεία και Ρύθμιση

Η υποδομή για το testing στο frontend έχει ήδη διαμορφωθεί και βασίζεται στα παρακάτω σύγχρονα εργαλεία:

- **Vitest:** Ένα σύγχρονο testing framework, πλήρως συμβατό με το Vite, που προσφέρει εξαιρετικά γρήγορη εκτέλεση των ελέγχων και ένα API παρόμοιο με το δημοφιλές Jest.
- **React Testing Library:** Μια βιβλιοθήκη που ενθαρρύνει τον έλεγχο των components με τον τρόπο που θα τα χρησιμοποιούσε ένας τελικός χρήστης, εστιάζοντας στο τι βλέπει και με τι μπορεί να αλληλεπιδράσει, αντί για τις εσωτερικές λεπτομέρειες υλοποίησης.
- **ESLint:** Αν και δεν είναι ένα testing framework, το ESLint παίζει κρίσιμο ρόλο στη διασφάλιση της ποιότητας του κώδικα. Έχει ρυθμιστεί (`eslint.config.js`) για να επιβάλλει κανόνες συνέπειας και να εντοπίζει πιθανά σφάλματα κατά τη φάση της ανάπτυξης, πριν καν εκτελεστούν οι έλεγχοι.

Η παραπάνω στοίβα εργαλείων έχει ενσωματωθεί στο project, παρέχοντας το απαραίτητο υπόβαθρο για τη συγγραφή και εκτέλεση αξιόπιστων ελέγχων.