ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών

WEB APPLICATION TECHNOLOGIES (ΥΣ14)

C# BackEnd | React FrontEnd

Dimitrios Fotopoulos
AM 1115202000292

Zisis Kammas
AM 1115202000290

Instructor:
Ioannis Chamodrakas

Athens, September 2024

Περιεχόμενα

1 Overview

This project is a professional networking web application designed to enable professionals manage their profiles, connect with other people with similar interests, and interact through articles, advertisements, and discussions. The application has two roles: **Administrator** and **Professional**, each with distinct features and interfaces.

Key Features

1. **User Roles:**
    - **Administrator:** Administrators can manage the users, roles, and export user information.
    - **Professional:** Professionals can manage their profiles, connect with other users, post and interact with articles, and apply for jobs.
2. **Professional Features:** Professionals have access to several functionalities:
    - Create and manage articles.
    - Post their professional experience, education, and skills.
    - Make and accept connection requests with other professionals.
    - Post and respond to job advertisements.
    - View and comment on articles posted by other professionals in their network.
    - Chat with their connections through private discussions..
    - Receive personalized job and article recommendations.
3. **Real-Time Communication:** Powered by **SignalR**, enabling real-time chat and notifications, including friend requests messages and article interactions.
4. **Personalized Recommendations:** The application supports a recommendation system using **Matrix Factorization**, to recommend job advertisements and articles on their feed, based on user interaction history.
5. **Backend:**
    - The backend is built in **C#**, handling business logic through controllers and services.
    - Models represent core data entities such as users, jobs, articles, advertisements, and connections.
6. **Frontend:**
    - The frontend is built with **React**, providing dynamic user interfaces for both administrators and professionals.

- The UI is responsive and role-based. Real-time updates and notifications are supported.

7. **Database:**
    - Stores user profiles, connections, connection requests, articles, jobs, messages, interactions, notifications, and the vectors used in the Matrix Factorization algorithm.
    - A database population script is provided to simulate real-world data for testing and development.

2 BackEnd

2.1 Models

Models designed for the database:

1. **User:** This model represents a user in the system and contains the following details: the unique identifier for each user (‘UserId‘), the user’s first name (‘FirstName‘), last name (‘LastName‘), email (‘Email‘), phone number (‘PhoneNumber‘), password (‘Password‘), date of birth (‘DateOfBirth‘), and address (‘Address‘). The model also includes the user’s profile picture, which is managed through two fields: ‘PhotoData‘ (for storing the image data) and ‘PhotoMimeType‘ (for storing the MIME type of the image). The ‘Admin‘ field indicates whether the user has administrative privileges.

   Additionally, this model has several relationships with other models:

    - A **one-to-many** relationship with the ‘Education‘ model, capturing the user’s educational background.
    - A **one-to-many** relationship with the ‘Job‘ model, capturing the user’s employment history.
    - A **one-to-many** relationship with the ‘Skill‘ model, capturing the user’s skills.
    - A **one-to-many** relationship with the ‘Message‘ model, representing messages sent by the user.
    - A **one-to-many** relationship with the ‘Advertisement‘ model, capturing advertisements created by the user.
    - A **one-to-many** relationship with the ‘AdvertisementVector‘ model, which captures the interaction vectors related to advertisements.
    - A **one-to-many** relationship with the ‘Article‘ model, representing articles authored by the user.

- A **one-to-many** relationship with the ‘Like‘ model, capturing the likes made by the user on various content.
- A **one-to-many** relationship with the ‘Comment‘ model, capturing the comments made by the user.

The ‘Network‘ property represents the user’s social network, containing a list of ‘Friendship‘ objects. The ‘PublicFields‘ property is a list of strings indicating which user fields are public.

2. **Education:** This model contains the following details: the unique identifier for each education entry (‘EducationId‘), the degree obtained (‘Degree‘), the level of education (‘Level‘), the name of the institution (‘Institution‘), the start and end dates (‘StartDate‘, ‘EndDate‘), the user to whom this education entry belongs (‘UserId‘), and a flag indicating whether the entry is public (‘IsPublic‘).

3. **Job:** This model contains the following details: the unique identifier for each job entry (‘JobId‘), the position held (‘Position‘), the industry of the job (‘Industry‘), the level of the job (‘Level‘), the name of the company (‘Company‘), the start and end dates (‘StartDate‘, ‘EndDate‘), the user to whom this job entry belongs (‘UserId‘), and a flag indicating whether the entry is public (‘IsPublic‘).

4. **Skill:** This model contains the following details: the unique identifier for each skill entry (‘SkillId‘), the name or category of the skill (‘SkillName‘), the proficiency level of the skill (‘Proficiency‘), the user to whom this skill entry belongs (‘UserId‘), and a flag indicating whether the entry is public (‘IsPublic‘).

5. **Friendship:** This model contains the following details: the unique identifier of the user (‘UserId‘), with a reference to the ‘User‘ object (‘User‘), the unique identifier of the friend (‘FriendId‘), with a reference to the ‘Friend‘ object (‘Friend‘), the date when the friendship was established (‘FriendshipDate‘), and a boolean property indicating whether the friendship request has been accepted (‘IsAccepted‘).

6. **ConnectionRequest:** This model contains the following details: the unique identifier for each connection request (‘Id‘), the identifier of the user who sent the request (‘SenderId‘), with a reference to the ‘User‘ object (‘Sender‘), the identifier of the user who received the request (‘ReceiverId‘), with a reference to the ‘User‘ object (‘Receiver‘), a boolean flag indicating whether the request has been accepted (‘IsAccepted‘), and the date when the connection request was sent (‘RequestDate‘).

7. **Discussion:** This model contains the following details: the unique identifier for each discussion (‘Id‘), the title of the discussion (‘Title‘), a list of participant user IDs (‘Participants‘), and a one-to-many relationship with messages, represented as a list of messages associated with the discussion (‘Messages‘).

8. **Message:** This model contains the following details: the unique identifier for each message (‘Id‘), the text content of the message (‘Text‘), the timestamp when the message was created (‘Timestamp‘), the unique identifier of the

sender (‘SenderId‘), the name of the sender (‘SenderName‘), the unique identifier of the discussion to which the message belongs (‘DiscussionId‘), and a list indicating the read statuses of the message by different users (‘ReadStatuses‘).

9. **MessageReadStatus:** This model contains the following details: the unique identifier for each message read status entry (‘Id‘), the identifier of the associated message (‘MessageId‘), the identifier of the user who read the message (‘UserId‘), and a boolean flag indicating whether the message has been read (‘IsRead‘).

10. **Advertisement:** This model contains the following details: the unique identifier for each advertisement (‘AdvertisementId‘), the title of the advertisement (‘Title‘), the description of the advertisement (‘Description‘), and the date the advertisement was posted (‘PostedDate‘). The model also includes minimum educational requirements, such as the required degree (‘RequiredDegree‘) and education level (‘RequiredEducationLevel‘), and minimum work experience requirements, such as the required job position (‘RequiredPosition‘), industry (‘RequiredIndustry‘), job level (‘RequiredJobLevel‘), and the minimum years of experience (‘MinimumYearsExperience‘). Additionally, the model specifies the required skill category (‘RequiredSkill‘). The model has a relationship with the user who posted the advertisement (‘UserId‘, ‘User‘), and it maintains a list of user IDs representing the applicants (‘ApplicantUserIds‘).

11. **AdvertisementVector:** This model contains the following details: the unique identifier for each advertisement vector (‘AdvertisementVectorId‘), the foreign key linking to the related advertisement (‘AdvertisementId‘), the required degree (‘RequiredDegree‘), the required education level (‘RequiredEducationLevel‘), the required job position (‘RequiredPosition‘), the required industry (‘RequiredIndustry‘), the required job level (‘RequiredJobLevel‘), the required skill (‘RequiredSkill‘), and the user to whom this vector belongs (‘UserId‘).

12. **Article:** This model contains the following details: the unique identifier for each article (‘ArticleId‘), the title of the article (‘Title‘), the content of the article (‘Content‘), and the date the article was posted (‘PostedDate‘). It also includes references to the author of the article (‘AuthorId‘, ‘Author‘). Additionally, the model supports multimedia content, including photos (‘PhotoData‘, ‘PhotoMimeType‘) and videos (‘VideoData‘, ‘VideoMimeType‘). The model maintains relationships with other models, including a collection of likes (‘Likes‘), a collection of comments (‘Comments‘) and a collection of views (‘Views‘).

13. **Like:** This model contains the following details: the unique identifier for each like (‘LikeId‘), the identifier of the user who liked the article (‘LikerId‘), with a reference to the ‘User‘ object (‘Liker‘), and the identifier of the liked article (‘ArticleId‘), with a reference to the ‘Article‘ object (‘Article‘).

14. **Comment:** This model contains the following details: the unique identifier for each comment (‘CommentId‘), the content of the comment (‘Content‘), the date the comment was posted (‘PostedDate‘), the identifier of the user who made the comment (‘CommenterId‘), with a reference to the ‘User‘ object (‘Commenter‘), and the identifier of the article being commented on (‘ArticleId‘), with a reference to the ‘Article‘ object (‘Article‘).

15. **View:** This model contains the following details: the unique identifier for each view (ViewId), the identifier of the user who viewed the article (ViewerId), with a reference to the User object (Viewer), and the identifier of the viewed article (ArticleId), with a reference to the Article object (Article). This model captures the relationship between users and the articles they have viewed, providing insight into user interactions with content.

16. **ArticleVector:** This model contains the following details: the unique identifier for each article vector (ArticleVectorId), the identifier of the article being interacted with (ArticleId), and a reference to the author of the article (AuthorId, Author). It also tracks the user who interacted with the article (UserId, User). Additionally, the model records the type of interaction (InteractionType), where the possible values are 1 for a like, 2 for a comment, and 3 for a view. This model is used to capture and categorize various interactions between users and articles.

17. **NoteOfInterest:** This model contains the following details: the unique identifier for each note of interest (Id), the identifier of the user who receives the notification (UserId), and a reference to the User object (User). It also includes the content of the notification (Content), the date and time the notification was created (CreatedAt), and a boolean field (IsRead) to track whether the notification has been read by the user. This model is used to manage notifications related to users’ activities and interactions (likes).

2.2 DTOs

Data Transfer Objects (DTOs) are used to encapsulate and transfer data between the server and client, reducing unnecessary data exposure and ensuring efficient communication. Below are the summaries of each DTO:

1. **ArticleDto:** This DTO is used to transfer summarized information about articles, including key fields like Title, Content, PostedDate, and AuthorName. It also includes counts of likes and lists of comments to reduce payload size while ensuring the client has relevant data without fetching all article-related details at once.

2. **CommentDto:** This DTO is used to transfer basic comment data, such as the comment’s content and the ‘CommenterName‘. It simplifies the comment structure for easy display without sending unnecessary user or article details.

3. **LikeDto:** This DTO is used to transfer the name of the user who liked an article (‘LikerName‘). It provides a lightweight way to send like-related data without exposing other fields from the ‘Like‘ model..
4. **ViewDto:** This DTO is used to transfer basic information about article views, specifically the ‘ViewerName‘. It helps track which users have viewed an article without exposing additional user or article details.
5. **AdvertisementDto:** This DTO is used to transfer key information about job advertisements, such as ‘Title‘, ‘Description‘, and qualification requirements (e.g., degree, experience). It summarizes the advertisement data for efficient display without loading all related models

2.3 Enums designed for the database

The enums defined in the MyApi.Models.Enums namespace are used to standardize the options available in the user registration process, particularly for filling out sections of a user’s profile such as education level, degrees, job industries, job levels, job positions, and skill categories. By using enums, these options are restricted to predefined choices, ensuring consistency in the data entered by users.When a user selects one of these standardized options during registration or profile setup, the corresponding value is stored in the database as an integer, which represents the selected enum value. This approach allows for more efficient storage and processing of user profile data, as well as ensuring that the data remains consistent and valid according to the defined options. This use of enums is particularly important in applications that require structured and easily queryable data, such as resumes or profiles, where certain fields must adhere to specific categories or levels, making the data both reliable and easier to work with in backend operations.

2.4 Controllers-Services

1. **UsersController - UserService:**
    - **GetAllUsers:** The GetAllUsers endpoint retrieves a list of all users from the database. It uses the UserService.GetAllUsers() method to fetch the data.The UserService.GetAllUsers() method queries the database to retrieve all users along with their related entities (Education, Jobs, Skills). The result is returned to the controller, which sends it back to the client as an HTTP response.
    - **GetUserById:** The GetUserById endpoint retrieves a specific user by their ID. It uses the UserService.GetUserById(int id) method to fetch the user. The UserService.GetUserById(int id) method queries the database for a user with the specified ID, including their Education, Jobs, and Skills relationships. If found, the user is returned to the controller, which then

sends it back to the client. If not found, a ”User not found” response is returned.

- **UpdateUser:** The UpdateUser endpoint updates the details of a specific user. It first checks if the provided ID matches the user ID in the request body, then uses the UserService.UpdateUser(User updatedUser) method to perform the update. The UserService.UpdateUser(User updatedUser) method retrieves the existing user from the database, updates their information, and saves the changes back to the database. If the user exists, their associated entities (Education, Jobs, Skills) are also updated. If the user does not exist, no action is taken.
- **DeleteUser:** The DeleteUser endpoint deletes a specific user by their ID. It uses the UserService.DeleteUser(int id) method to perform the deletion. The UserService.DeleteUser(int id) method finds the user by their ID and removes them from the database. If the user exists, they are deleted; otherwise, no action is taken.
- **GetUserNamesById:** The GetUserNamesById endpoint retrieves the first and last names of a specific user by their ID. It uses the UserService.GetUserNamesById(int id) method to perform this retrieval. The UserService.GetUserNamesById(int id) method queries the database for the user with the specified ID and returns their first and last names. If the user is found, their names are returned; if not, null is returned, leading to a ”Not Found” response from the controller.

2. **UserLoginController - UserService:** The Login endpoint in the UserLoginController handles user login requests. It validates the email and password provided in the LoginRequest body, checks them against the stored user data, and returns a response based on whether the credentials are correct. The controller calls the UserService.GetUserByEmailAndPassword(string email, string password) method, which queries the database to find a user matching the provided email and password. **LoginRequest Model:** The LoginRequest model is used to encapsulate the data sent in the login request. It contains the Email and Password fields that the user provides to log in (DTO use).

3. **UserSettingsController - UserSettingsService:**

   - **UpdateEmail:** The UpdateEmail endpoint allows a user to update their email address. It first validates the email, then calls the UserSettingsService.UpdateEmailAsync(int userId, string email) method to perform the update.If all checks pass, the service updates the user’s email and saves the changes to the database. The controller then returns a NoContent response, indicating that the update was successful.
   - **UpdatePassword:** The UpdatePassword endpoint allows a user to update their password. It validates the password and then calls the UserSettingsService.UpdatePasswordAsync(int userId, string password) method to perform the update. The controller checks if the password

provided in the request body is not null or empty. If it is, a BadRequest response is returned. The service method checks if the user exists in the database. If the user is not found, a ServiceResponse indicating failure is returned.If the user is found, the service updates the user's password and saves the changes to the database. The controller then returns a NoContent response, indicating that the update was successful.

- **UpdateEmailRequest - UpdatePasswordRequest Models:** These models are used to encapsulate the data sent in the update email and update password requests, respectively. They contain the Email and Password fields that the user provides (DTO use).
- **ServiceResponse Model:** The ServiceResponse class is used to encapsulate the result of a service operation, indicating whether it was successful and providing a message if needed (DTO use).

4. **UserRegistrationController - UserRegistrationService:** The Register endpoint in the UserRegistrationController handles user registration requests. It validates the necessary fields (first name, last name, email, password, and date of birth), checks if the email is already in use, processes the photo and public fields, and then creates a new User object to save in the database. The controller uses the UserRegistrationService.IsEmailInUse(email) method to check if the email provided in the registration request is already associated with an existing user. If the email is in use, it returns a BadRequest response with a corresponding message. If the email is not in use, the controller processes the optional photo and public fields, constructs a new User object, and passes it to the UserRegistrationService.AddUser(user) method to save the user to the database. If the user is successfully saved, the controller returns a 200 OK response with the user's ID and email. If an error occurs during saving, a 500 Internal Server Error response is returned. **UserRegistrationRequest Model:** The UserRegistrationRequest model is used to encapsulate the data sent in the user registration request. It contains the user's first name, last name, email, phone number, password, optional photo, date of birth, address, and public fields.

5. **UserBasicInfoController - UserBasicInfoService:** The UpdateBasicInfo endpoint in the UserBasicInfoController handles requests to update a user's basic information, including their phone number, address, and optional profile photo. The controller logs the receipt of the update request and then calls the UserBasicInfoService.UpdateUserBasicInfo(int userId, string phoneNumber, string address, IFormFile? photo) method to perform the update. The service method retrieves the user by their userId. If the user is found, it updates the phone number and address. If a photo is provided, it processes the photo, converting it into a byte array, and saves both the photo data and its MIME type to the user's record. The service saves the updated user information back to the database. If any errors occur during this process, the controller catches the exception, logs it, and returns a 500 Internal Server

Error response. If successful, it returns a 200 OK response with a confirmation message.

6. **UserBasicInfoController - UserBasicInfoService:** The ExportUsers endpoint in the UserExportController handles the export of user data based on a list of user IDs provided in the request body. The format for the export (either JSON or XML) is specified in the query string. The controller first retrieves the users based on the provided user IDs by calling UserExportService.GetUsersByIds(List<int> userIds). Depending on the format specified in the query string (json or xml), the controller serializes the list of users into the appropriate format. For JSON, the controller uses the Newtonsoft.Json.JsonConvert.SerializeObject method with settings to properly format enums and the JSON output. For XML, the controller uses the XmlSerializer to serialize the user data to XML format. The serialized data is then returned as a file in the appropriate format. If an invalid format is specified, the controller returns a BadRequest response.

7. **EnumController - EnumService:** The GetAllEnums endpoint in the EnumController provides a way to retrieve all the enum values used within the application. It returns a dictionary where the keys are the names of the enums and the values are lists of the corresponding enum names. The controller calls the EnumService.GetAllEnums() method to get a dictionary containing all the enum names and their values. The service method uses Enum.GetNames to retrieve the names of each enum type, which it stores in a dictionary. This dictionary is then returned to the controller, which sends it back as an OK (HTTP 200) response containing the dictionary.

8. **DiscussionsController - DiscussionService:**

    - **CreateDiscussion:** CreateDiscussion endpoint allows the creation of a new discussion between users. It ensures that at least two participants are provided and checks whether a discussion with the same participants already exists. If not, it creates a new discussion and returns it. The service checks for an existing discussion with the same participants, generates a title from the participants' names, and saves the new discussion in the database. If a similar discussion already exists, it returns that discussion instead of creating a new one.
    - **GetDiscussion:** The GetDiscussion endpoint retrieves a discussion by its ID, including all related messages. The service method loads the discussion from the database, including its associated messages, and returns it to the controller.
    - **GetDiscussionsByUser:** The GetDiscussionsByUser endpoint retrieves all discussions for a specific user, including the count of unread messages. The service loads discussions involving the user, calculates the number of unread messages for that user in each discussion, and returns this information to the controller.

- **AddParticipant:** The AddParticipant endpoint allows adding a new participant to an existing discussion. If the discussion does not exist or the user is already a participant, it returns an error. The service method adds the user to the participants list and updates the discussion title to include the new participant's name, then saves the changes to the database.
- **RemoveParticipant:** The RemoveParticipant endpoint allows removing a participant from an existing discussion. If the discussion does not exist or the user is not a participant, it returns an error. The service method removes the user from the participants list and updates the discussion title to remove the participant's name, then saves the changes to the database.
- **DeleteDiscussionOrRemoveParticipant:** The DeleteDiscussionOrRemoveParticipant endpoint either deletes a discussion if there are only two participants, or removes a specific participant if there are more than two. This ensures that discussions with only two participants are removed once one leaves, while discussions with more participants continue with the remaining members. The service method checks if the discussion exists and whether the user is a participant. If the discussion only has two participants, it deletes the entire discussion. Otherwise, it removes the participant from the discussion and updates the discussion title accordingly before saving the changes to the database.

9. **MessagesController - MessageService:**

    - **SendMessage:** The SendMessage endpoint handles sending a new message within a discussion. If the discussion does not exist, it creates a new one. The message is then sent to all participants of the discussion using SignalR for real-time communication. The MessageService.SendMessage method validates the sender and the discussion, assigns the sender's name to the message, and saves the message to the database. The method then sends the message to all connected clients via SignalR.
    - **MarkMessagesAsRead:** The MarkMessagesAsRead endpoint marks all messages in a discussion as read for a specific user. The MessageService.MarkMessagesAsRead method retrieves all messages in a discussion, updates the read status for the specified user, and saves the changes if any messages were updated.

10. **AdvertisementController - AdvertisementService:**

    - **CreateAdvertisement:** The CreateAdvertisement endpoint allows users to create a new advertisement. The service method associates the advertisement with the user who created it, saves it in the database, and returns the created advertisement. The service method retrieves the user, adds the new advertisement to the user's list, and saves the changes to the database.

- **UpdateAdvertisement:** The UpdateAdvertisement endpoint allows users to update an existing advertisement's details. The service method retrieves the advertisement, updates its details with the provided information, and saves the changes to the database.
- **DeleteAdvertisement:** The DeleteAdvertisement endpoint allows a user to delete an advertisement they created. The service method verifies the user's ownership of the advertisement, deletes it from the database, and saves the changes.
- **GetAdvertisement:** The GetAdvertisement endpoint retrieves an advertisement by its ID. The service method finds the advertisement in the database and returns it to the controller.
- **GetAllAdvertisements:** The GetAllAdvertisements endpoint retrieves all advertisements. The service method returns a list of all advertisements from the database.
- **GetAdvertisementsByUser:** The GetAdvertisementsByUser endpoint retrieves all advertisements created by a specific user. The service method finds all advertisements in the database that were created by the specified user.
- **GetParticipantsByAdvertisementId:** The GetParticipantsByAdvertisementId endpoint retrieves all participants (users who applied) for a specific advertisement. The service method finds the advertisement, retrieves the list of participants, and returns it.
- **RemoveParticipant:** The RemoveParticipant endpoint removes a participant from an advertisement's list of applicants. The service method verifies the participant's existence and removes them from the advertisement's list of applicants before saving the changes.
- **GetFilteredAdvertisements:** The GetFilteredAdvertisements endpoint retrieves advertisements that match a user's qualifications, such as education and job experience. The service method filters advertisements based on the user's highest degree, education level, job industry, and job level. It returns a list of advertisements that closely match the user's profile.
- **AddParticipant:** he AddParticipant endpoint allows a user to apply to an advertisement. If the participant is already associated with the advertisement, an error is returned. The service method checks if the participant has already applied. If not, it adds the participant's ID to the list of applicants for the advertisement and saves the changes to the database.

11. **AdvertisementVectorController - AdvertisementVectorService:**

    - **AddAdvertisementVector:** The AddAdvertisementVector endpoint allows the creation of a new advertisement vector, which is a representation of user interactions or preferences related to job advertisements. If the user exists, the vector is added to their interaction

vectors list.The service method retrieves the user, adds the new advertisement vector to the user's list of interaction vectors, and saves the changes to the database. If the user does not exist, it returns false.

- **GetRecommendedAdvertisements:** The GetRecommendedAdvertisements endpoint provides job advertisement recommendations for a user based on their interaction vectors (previous interactions or preferences). The service method uses the user's interaction vectors and compares them with the attributes of available advertisements to compute similarity scores. It then applies **Matrix Factorization** to predict and rank the advertisements, returning the top recommendations.

12. **ArticleController - ArticleService :**

    - **GetAllArticles:** The GetAllArticles endpoint retrieves all articles in the database and returns them in a structured DTO format that includes information about the article, its author, likes, and comments. The service method fetches all articles, including their related data (author, likes, comments), and maps them into a list of ArticleDto objects before returning them to the controller.
    - **GetArticleById:** The GetArticleById endpoint retrieves a single article by its ID and returns it in a detailed DTO format. The service method looks up the article by its ID, includes its related data (author, likes, comments), and returns it as an ArticleDto.
    - **CreateArticle:** The CreateArticle endpoint allows for the creation of a new article, including optional photo and video content. It takes form data and files, processes them, and saves the article. The service method adds the new article to the database and saves it. The controller then returns the created article in a DTO format.
    - **LikeArticle:** The LikeArticle endpoint allows a user to like an article. If the user has already liked the article, the request is ignored. The service method checks if the article exists and if the user has already liked it. If not, it adds a like, creates a notification for the author, and sends a SignalR notification.
    - **UnlikeArticle:** The UnlikeArticle endpoint allows a user to remove their like from an article. The service method checks if the like exists in the database. If it does, the like is removed, and the changes are saved.
    - **GetLikedArticles:** The GetLikedArticles endpoint retrieves a list of article IDs that a specific user has liked. The service method queries the Likes table to find all articles liked by the user and returns the list of article IDs.
    - **AddComment:** The AddComment endpoint allows users to add a comment to an article. The service method checks if the article exists. If it does, the comment is associated with the article and saved to the database.

13. **ArticleVectorController - ArticleVectorService:**

    - **AddArticleVector:** The AddArticleVector endpoint allows the creation of a new article interaction vector (representing a like, comment, or view) for a specific article and user. If the user exists, the service method adds the new vector to their list of article interaction vectors and saves it to the database. If the user does not exist, the method returns false.
    - **RemoveArticleVector:** The RemoveArticleVector endpoint allows the removal of an existing article interaction vector (such as a like). The service method removes the vector from the database if it exists and clears the associated user's cached article recommendations to ensure up-to-date results in the future.
    - **GetRecommendedArticles:** The GetRecommendedArticles endpoint provides personalized article recommendations for a user based on their interaction vectors (likes, comments, views). The service method retrieves the user's interaction history and uses a combination of similarity scoring, time decay for older articles, and **Matrix Factorization** to rank and return the most relevant articles for the user.

2.5 ChatHub

The ChatHub class inherits from Hub, which is part of the SignalR library. SignalR allows for real-time web functionality, enabling server-side code to push content to connected clients instantly as it becomes available.

- **SendMessage:** This method sends a message from a user to all participants in a discussion except the sender. It loops through the list of participant IDs, sending the message to each user.This method is typically used in a chat application where a user sends a message to a group discussion. The message is broadcasted to all participants, and the sender receives a confirmation that the message has been sent.
- **SendFriendRequestNotification:** This method sends a friend request notification to a specific user. When a user sends a friend request to another user, this method is triggered to notify the recipient in real-time that they have received a friend request.
- **SendNoteOfInterestNotification:** This method sends a notification related to a note of interest to a specific user.
- **PingHub:** This method is a simple ping-pong test to check if the hub is working correctly.

2.6 AppDBContext

The AppDbContext class inherits from DbContext, which is a part of Entity Framework Core (EF Core). This class is responsible for managing the database operations related to the different entities (models) in your application. It defines the structure of the database through DbSet properties and configures relationships between entities using the OnModelCreating method.The AppDbContext class is crucial for interacting with your database, defining how your models are structured in the database, and configuring relationships between different entities. It ensures that the database schema aligns with the application's data models and enforces data integrity through relationships and constraints.

2.7 Utilities (Matrix Factorization)

The MatrixFactorization class is a utility used by both the ArticleVectorService and AdvertisementVectorService to generate personalized recommendations by learning latent features from user interactions with articles or advertisements. It implements collaborative filtering through matrix factorization, predicting user preferences based on interaction history.

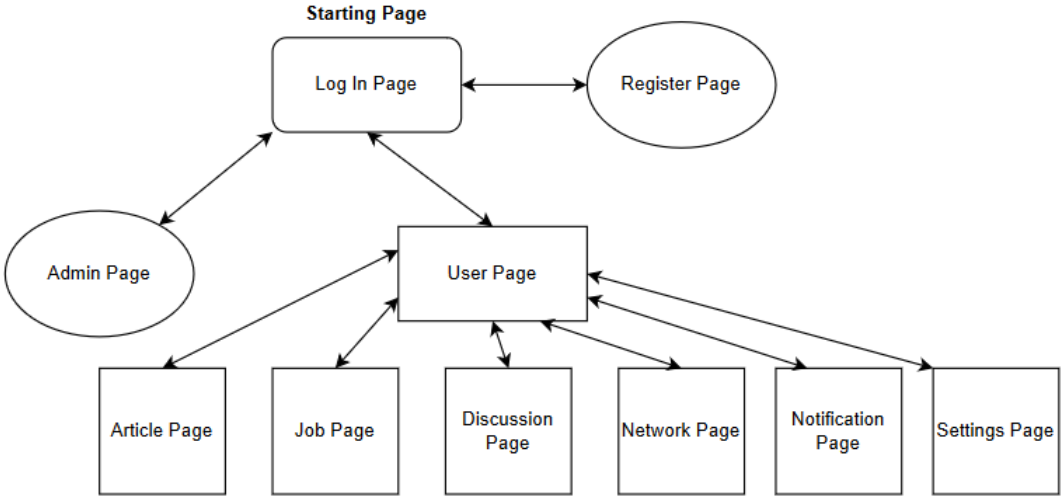
- **Initialization:** The MatrixFactorization class is initialized with the following parameters:
  - **numUsers:** The number of users in the dataset.
  - **numItems:** The number of items (articles or advertisements) in the dataset.
  - **numLatentFeatures:** The number of latent features used to model user-item interactions.
  - **learningRate:** The step size used for updating the feature matrices during training.
  - **regularization:** A parameter that controls the complexity of the model to prevent overfitting.
  - **numIterations:** The number of iterations for the training process.
- **Training:** The 'Train' method is used to learn the latent features from a user-item interaction matrix (e.g., ratings, likes, views). It iteratively updates the user and item matrices by minimizing the prediction error between the actual interaction values and the predicted ones, while applying regularization to control model complexity.
- **Prediction:** The 'PredictRating' method computes the predicted interaction value (such as a like, view, or comment) between a user and an item by multiplying the user's latent features with the item's latent features. This is used to generate recommendations for the user.

- **GetPredictedRatings:** This method returns the full matrix of predicted ratings, where each cell represents the predicted interaction between a user and an item. It is used to rank items (articles or advertisements) and select the top recommendations for a user.

Matrix factorization provides an efficient and scalable way to predict user preferences, making it ideal for generating personalized content in the application.

2.8 WebApplication Configuration - Program.cs

The code provided is used to configure a .NET Core web application, specifically setting up services, middleware, and the HTTP request pipeline. Configures the AppDbContext to use SQLite as the database, with the file name database.db. This service is added to the dependency injection (DI) container and will be used to interact with the database.Registers various services for DI. These services handle business logic related to users, registrations, bios, exports, networks, settings, messages, discussions, enums, advertisements, vectors, and articles. Adds SignalR to the project, which allows for real-time web functionality (like chat). Adds support for MVC controllers, enabling the application to handle HTTP requests via controllers. Configures a CORS policy named AllowReactApp. This policy allows requests from http://localhost:3000 and https://localhost:3000, which are typical development addresses for a React frontend. It allows any HTTP method, any header, and credentials (cookies, HTTP authentication, etc.). Configures Kestrel, the web server used by .NET Core, to listen on port 5297 for HTTP traffic and on port 7176 for HTTPS traffic. Builds the web application with the configured services and settings. In the development environment, it enables a developer-friendly exception page. Enables routing for the application, allowing it to match requests to the appropriate endpoints. Applies the previously defined CORS policy, ensuring that only allowed origins can make requests to the API. Redirects HTTP requests to HTTPS to ensure secure communication. Ensures that authorization is enforced for the API endpoints. Maps the ChatHub to the /chatHub endpoint, enabling real-time communication via SignalR for chat functionality. Maps the controllers in the application, making them available to handle incoming HTTP requests. Starts the application, listening for incoming requests.



3 FrontEnd

3.1 Flowchart

3.2 Header - Footer

Header dynamically updates based on the user's role and session status, providing navigation, real-time updates, and notification handling in a structured and responsive manner.

Footer component dynamically changes its message based on the user's login status and role.

3.3 Log In Page - (Home.js)

1. **State Management**: Manages email, password, error messages, user name, and modal visibility.
2. **Login Functionality**: Submits login details, displays a success modal with the user's name, and redirects based on user role (admin or user).
3. **Password Reclaim**: Allows users to reclaim their password by entering their email, which triggers an API request to send reclaim instructions.
4. **UI Elements**: Includes an introductory section highlighting the app's benefits, a login form with validation, and success/error modals for login and password reclaim events.

3.4 Register Page - (Register.js and RegisterBio.js)

Register

1. **State Management**: Utilizes the `useState` hook to manage user data (e.g., name, email, password) and modal visibility for error and success messages.
2. **Profile Photo Handling**: Allows users to upload a profile picture or, if none is provided, selects a random image from a predefined set of stock images.
3. **Form Submission**: Validates the form (e.g., password confirmation), collects public/private user field settings, and submits the data as `FormData` via an API request to the backend.
4. **Conditional Modals**: Displays success or error modals based on the registration outcome, including a modal for email uniqueness and one for completing the first step of registration.
5. **Navigation**: On successful registration, it navigates the user to the biography writing page or allows them to skip to their user dashboard.

RegisterBio

1. **State Management**: Manages state for education, job, skills, and the fetched enums using `useState`.
2. **Education Section**: Allows users to input education details such as degree, institution, start and end dates. Provides functions to add, edit, and delete education entries.
3. **Job Section**: Users can input job details including position, company, and work period. Similar to education, entries can be added, edited, or deleted.
4. **Skill Section**: Enables the input of skills and proficiency, with options to add, edit, or delete skills.
5. **Form Submission**: Submits the user's bio data (including education, job, and skills) to the backend. Dates are formatted to ISO format before submission.
6. **Modals for Feedback**: Displays a success modal when the biography is created, providing feedback to the user.
7. **Error Handling**: Validates input fields such as date ranges and ensures mandatory fields are filled, displaying error messages when necessary.

3.5 Admin Page - (Admin.js ,EditUserModal.js and Peginitation Component.js)

1. **State Management**: Utilizes `useState` to manage the state of users, selected users, pagination, and modals for session timeouts and editing users.
2. **Session Timeout**: Implements an idle timer that logs out users after 5 minutes of inactivity, displaying a warning modal before logging out.
3. **User Management**: Fetches a list of all users from an API and allows for selecting, editing, and deleting users. Users can be filtered into different categories such as admins and non-admins, with the ability to select all or individual users.
4. **Export Functionality**: Allows the admin to export user data in either JSON or XML format for the selected users.
5. **Pagination**: Includes pagination functionality to control the number of users displayed per page, with configurable options for 50, 100, or 200 rows per page.
6. **User Editing**: Provides a modal to edit user details, including education, jobs, and skills, and handles validation for date ranges.
7. **Modals for Feedback**: Displays modals for session timeout warnings and user editing.

3.6 User Page -(User.js,CreateArticle.js and UserArticle.js)

1. **User Component**:
   - Provides a user dashboard with navigation links for ”Profile” and ”Network”.
   - Integrates session management with an idle timer that logs the user out after 5 minutes of inactivity, showing a warning modal.
   - Contains both the `CreateArticle` component for article creation and the `UserArticles` component for displaying a list of articles.
2. **CreateArticle Component**:
   - Allows users to create new articles by submitting a form with a title, content, and optional photo or video.
   - Utilizes `axios` to send a `POST` request to the server and handles the article submission with a loading spinner.
   - Shows success feedback when an article is created successfully and clears the form afterward.

3. **UserArticles Component**:
    - Fetches and displays a list of articles, either recommended or regular, for the logged-in user.
    - Handles user interactions, such as liking/unliking articles and posting comments.
    - Supports infinite scrolling with a ”Load more” button to load additional articles dynamically.
    - Displays multimedia content (photos/videos) within articles and includes modals for viewing images and detailed articles.
    - Allows users to view comments on articles and post new comments in real-time.

3.7 Discussion Page (UserDiscussion.js)

1. **State Management**:
    - Manages discussions, messages, friends, and participants using `useState`.
    - Utilizes `useContext` to access user information, unread messages, and selected discussion data.
    - Handles real-time message reception using the `SignalRContext`.
2. **Fetching Data**:
    - Fetches discussions for the current user and displays them in a list.
    - Retrieves messages for the selected discussion and marks messages as read.
    - Fetches a list of the user’s friends and allows adding them to discussions.
3. **Messaging and Discussion Management**:
    - Users can send messages in a selected discussion, and real-time updates are reflected immediately in the UI.
    - Provides options to add or remove participants from discussions, with modal dialogs for selecting participants.
    - Allows the creator of a discussion to delete it or remove participants.
4. **Real-Time Updates**:
    - Listens for new messages via the `SignalRContext`, updating the discussion’s message list and unread count dynamically.
    - Scrolls to the bottom of the message list automatically when new messages are received or sent.
5. **Modals for Interaction**:
    - Includes modals for adding or removing participants, with friend and participant selection.

3.8 Network Page (UserNetwork.js and UserNetworkProfile.js)

1. **UserNetwork Component**:
    - Displays the current user's friends and allows them to search for other users in the network.
    - Includes actions to view user profiles, start a chat, or remove/block friends.
    - Provides search functionality with paginated results.
    - Integrates a chat modal for sending messages directly to selected friends.
    - Uses `axios` for API calls to fetch friends, send friend requests, and initiate discussions.
2. **UserNetworkProfile Component**:
    - Displays detailed information about a specific user's profile, including public education, job, and skill details.
    - Includes an option to initiate a chat with the profile owner through a modal dialog.
    - Fetches public user details, including their education, jobs, and skills, and presents them in a formatted list.
    - Leverages API calls to retrieve user data and initiate discussions.

3.9 Notification Page (UserNotification.js)

1. **Purpose**:
    - Displays the user's notifications, including friend connection requests and notes of interest.
    - Allows the user to accept or reject connection requests.
2. **Key Features**:
    - **Tabs**: Provides two tabs:
        - **Notes of Interest Tab**: Shows unread notes and marks them as read when selected.
        - **Connection Requests Tab**: Lists pending connection requests, allowing the user to accept or reject them.
    - **Badge Indicators**: Displays badges with the number of unread notifications for each tab.
    - **API Calls**: Fetches notifications using `axios` and updates the state after the user interacts with the notifications.

- **Error Handling**: Handles and displays errors if fetching notifications or processing requests fails.

3. **Context Integration**:
   - Uses the `UserContext` to access the current user's data.
   - Uses the `SignalRContext` to handle real-time updates and reset unread notifications count.

3.10 Job Page (UserJobs.js - UserJobsCreate.js - UserJobsView - UserAdParticipants)

UserJobs.js

1. **Purpose**: Displays job advertisements for the user and allows CV submission to employers.
2. **Key Features**: Fetches recommended or general job ads, displays job details such as required degree, industry, and experience, and allows users to send CVs with a confirmation modal.
3. **Context Integration**: Uses `UserContext` for accessing the current user's data.
4. **User Interface**: Includes a sidebar with job management options and a card layout for displaying job ads.

UserJobsCreate.js

1. **Purpose**: Provides a form for users to create job advertisements.
2. **Key Fields**: Job title, description, degree, education level, position, industry, job level, experience, and required skill.
3. **Data Fetching**: Retrieves enum options for dropdown fields from the backend.
4. **Form Submission**: Sends a POST request to create the job ad and handles errors.
5. **Navigation**: Redirects to "View Advertisements" upon success.

UserJobsView.js

1. **Purpose**: Displays job advertisements created by the user and allows for viewing, editing, and deleting ads.
2. **Key Features**:

- Fetches a list of the user's job ads.
- Provides a form to view and edit the selected job ad details.
- Allows deleting the selected job ad with a confirmation modal.

3. **Form Functionality**: Allows editing fields such as job title, description, degree, education level, job position, industry, job level, required skill, and minimum years of experience.
4. **API Calls**: Handles GET requests to fetch job data and enums, PUT requests to update ads, and DELETE requests to remove ads.
5. **Modals**:
    - Success modal for confirming the update.
    - Delete confirmation modal for removing the ad.

UserAdParticipants.js

1. **Purpose**: Manage job advertisements and participants by approving or dismissing applicants.
2. **Key Features**:
    - Displays user-created job ads and participant details.
    - Allows approving or dismissing participants.
    - Fetches data via `axios`.
    - Uses modals for confirmation messages.
3. **Context**: Accesses user data from `UserContext`.

4 Database Population Script

4.1 Overview

The Database Population Script is a utility designed to populate the database with sample data for users, connections, articles, and interactions. It simulates real-world data that can be used to test and develop features for the application. In addition, it serves to make the **Matrix Factorization** algorithm work with a sufficient amount of data. The script is configurable, allowing the creation of a variable number of users, connections, articles, and interactions.

4.2 Script Configuration

The script allows for the configuration of several parameters that control the amount and variety of data generated:

- **userCount:** The total number of users to generate. By default, this is set to 100.
- **leastConnectionCount** / **mostConnectionCount:** The minimum and maximum number of connections (friendships) each user should have. The script randomly assigns a number of connections within this range.
- **leastArticleCount** / **mostArticleCount:** The minimum and maximum number of articles each user should create.
- **leastLikeCount** / **mostLikeCount:** The minimum and maximum number of likes each user should give.
- **leastCommentCount** / **mostCommentCount:** The minimum and maximum number of comments each user should post.

4.3 User Creation

The script creates a variety of users with different profiles, educational backgrounds, job histories, and skills. Here's a detailed breakdown:

- **Admin User:** A special user is created with administrative privileges. This user has predefined details such as a fixed email (admin@example.com), and basic profile data.
- **Regular Users:**
  - **Personal Details:** Each user is generated with a first name, last name, email, phone number, password, date of birth, and address.
  - **Education:** Users may have educational histories that include high school, bachelor's, master's, and doctorate degrees. The script uses predefined lists of universities and degree types to generate this data.
  - **Jobs:** The script generates a job history for each user, mapping their degrees to likely job positions and industries. It considers career progression, with users potentially advancing from internships to senior-level positions.
  - **Skills:** Users are assigned random skills from a predefined list, with each skill associated with a proficiency level.
  - **Profile Pictures:** Users are assigned profile pictures randomly selected from a set of stock images.

- **Advertisements:** If a user's job history indicates senior-level positions, the script generates advertisements related to their last job position and industry.

4.4 Connection Generation

The script creates a network of connections (friendships) between users:

- **Friendship Creation:** For each user, a number of friendships are created based on the specified range (leastConnectionCount to mostConnectionCount). Friendships are preferably formed between users with common industries, degrees, or skills.
- **Automatic Acceptance:** All connection requests generated by the script are automatically accepted.

4.5 Article Generation

Each user is assigned a number of articles:

- **Article Content:** Articles are generated with titles and content that are contextually relevant to the user's job history or educational background.
- **Publication Date:** The publication dates of articles are randomized within the last year.

4.6 Interaction Generation

The script simulates user interactions with articles:

- **Likes:** Users "like" a specified number of articles, with a preference for articles written by their connections.
- **Comments:** Users also comment on articles. Comments are generated from a list of predefined phrases.
- **Article Vectors:** For each interaction (like or comment), a vector is created to represent the interaction type, the article, and the user involved. These vectors are necessary for the Matrix Factorization algorithm used to implement the article recommendation system.

4.7 Execution

To run the script, inside the folder ./Scripts/UserCreationScriptApp type:

```
dotnet run [userCount]
```

Where [userCount] is an optional argument specifying the number of users to generate. For the Matrix Factorization to work, 1000 users are enough. If not provided, the script defaults to creating 100 users.

4.8 Customization

The script can be modified to adjust the ranges and probabilities for generating various user attributes, job histories, educational backgrounds, and interactions. This flexibility allows developers to simulate different data scenarios and test the application's features under varied conditions.

5 Difficulties

The main difficulties we encountered in this project are:

1. The interface in discussions, messages, notifications, and friend requests should work in real-time. For example, in a discussion, the messages should be sent directly to the other user(s) without the need for a page refresh.
2. To create a script that generates users with appropriate and realistic associations, so that the application and corresponding MF algorithms can work with realistic data.
3. To categorize the user based on certain aspects of their education, work experience, and skills. The method we chose involves some standard options that were initialized using an enum.
4. To integrate the MF (Jobs/articles) so that it works for a single user with the corresponding table, and not for all users, ensuring good time complexity. This allows the algorithm to run automatically when the corresponding page is opened, rather than as a backend technical job.
5. To create a realistic initial rating for the MF based on the corresponding vectors we generated for jobs and articles, and to address the problems that arose during the process.

6 GitHub

- Zisis Kammas GitHub
- Dimitris Fotopoulos GitHub