Documentatie

**UAPP**

Echipa: Tosa Cezar Dumitru, Ricu Alexandru Razvan, Moga Eduard Mihai, Stan Mihai Alexandru

**Descrierea Proiectului**

**UApp** este o aplicație web dezvoltată utilizând o arhitectură client-server.

* **Frontend:** Dezvoltat în **Next.js**, un framework React pentru crearea de aplicații web de înaltă performanță și cu randare server-side (SSR).
* **Backend:** Implementat în **Django**, un framework Python robust pentru dezvoltarea de aplicații web, oferind un sistem de ORM (Object-Relational Mapper) puternic pentru interacțiunea cu baza de date.
* **Baza de date:**
  + **MySQL:** Utilizată pentru stocarea datelor principale ale aplicației, cum ar fi informațiile utilizatorilor, rezervările, anunțurile și alte entități relevante.
  + **Firebase:** Integrată pentru a facilita comunicarea în timp real în cadrul chat-ului aplicației, oferind funcționalități precum trimiterea și primirea de mesaje instantanee și prezența utilizatorilor.

**Funcționalități Principale:**

* **Autentificare Utilizatori:**
  + Înregistrare și autentificare utilizatori prin intermediul frontend-ului Next.js.
  + Gestionarea sesiunilor utilizator și implementarea mecanismelor de securitate adecvate.
* **Forum de Discuții:**
  + Crearea, vizualizarea și comentarea topicurilor de discuție.
  + Sistem de filtrare și căutare pentru a găsi cu ușurință informații relevante.
* **Chat în Timp Real:**
  + Implementarea chat-ului utilizând Firebase, permițând comunicarea instantanee între utilizatori.
* **Rezervări Online:**
  + Sistem de rezervare pentru mașini de spălat și locuri de parcare.
  + Gestionarea disponibilității resurselor și a rezervărilor efectuate.
* **Panel de Administrare (Admin):**
  + Permisiuni de acces restricționate pentru administratori.
  + Gestionarea utilizatorilor, anunțurilor, rezervărilor și altor aspecte administrative.
  + Aprobare/declinare a cererilor de rezervare pentru locuri de parcare.

**Arhitectură:**

* **Client (Frontend):** Next.js se ocupă de renderizarea interfeței utilizator, gestionarea stării aplicației și interacțiunea cu API-ul backend.
* **Server (Backend):** Django gestionează logica de afaceri, interacționează cu baza de date MySQL și expune un API RESTful pentru a comunica cu frontend-ul.
* **Baza de Date:** MySQL stochează majoritatea datelor aplicației, în timp ce Firebase este utilizat pentru comunicarea în timp real în cadrul chat-ului.

**Tehnologii Utilizate:**

* **Frontend:** Next.js, React, JavaScript, CSS/SCSS
* **Backend:** Django, Python, Django REST Framework
* **Baza de Date:** MySQL, Firebase
* **Control de Versiune:** Git

A diagram of a person

Description automatically generated

**Tosa Dumitru Cezar:**

Eu am facut urmatoarele diagrame:

1. Pentru functionalitatea de washing machine am realizat activity diagram:

A screenshot of a computer flowchart

Description automatically generated

1. Pentru functionalitatea de washing machine am realizat state diagram:

A screenshot of a computer flowchart

Description automatically generated

Ca design pattern am ales Mediator pattern pentru functionalitatea de parking.

* Constructorul pentru clasa mediator care contine un service pentru callurile la api si un handler pentru UI

A computer screen with text

Description automatically generated

* Service este folosit pentru callurile la api, continand metode abstracte pe care le poate injecta clasa mediator precum:

A screen shot of a computer screen

Description automatically generated

* UIHandler este folosit pentru a updata UI-ul si pentru fiecare api call, se va apela o metoda specifica prin care vom updata UI-ul. De asemenea, metodele respective sunt date in constructorul clasei pentru ca fiecare pagina poate sa foloseasca altfel resursele primite.

A screen shot of a computer program

Description automatically generated

* Also we have a singleton object of type mediator that we use across our app:

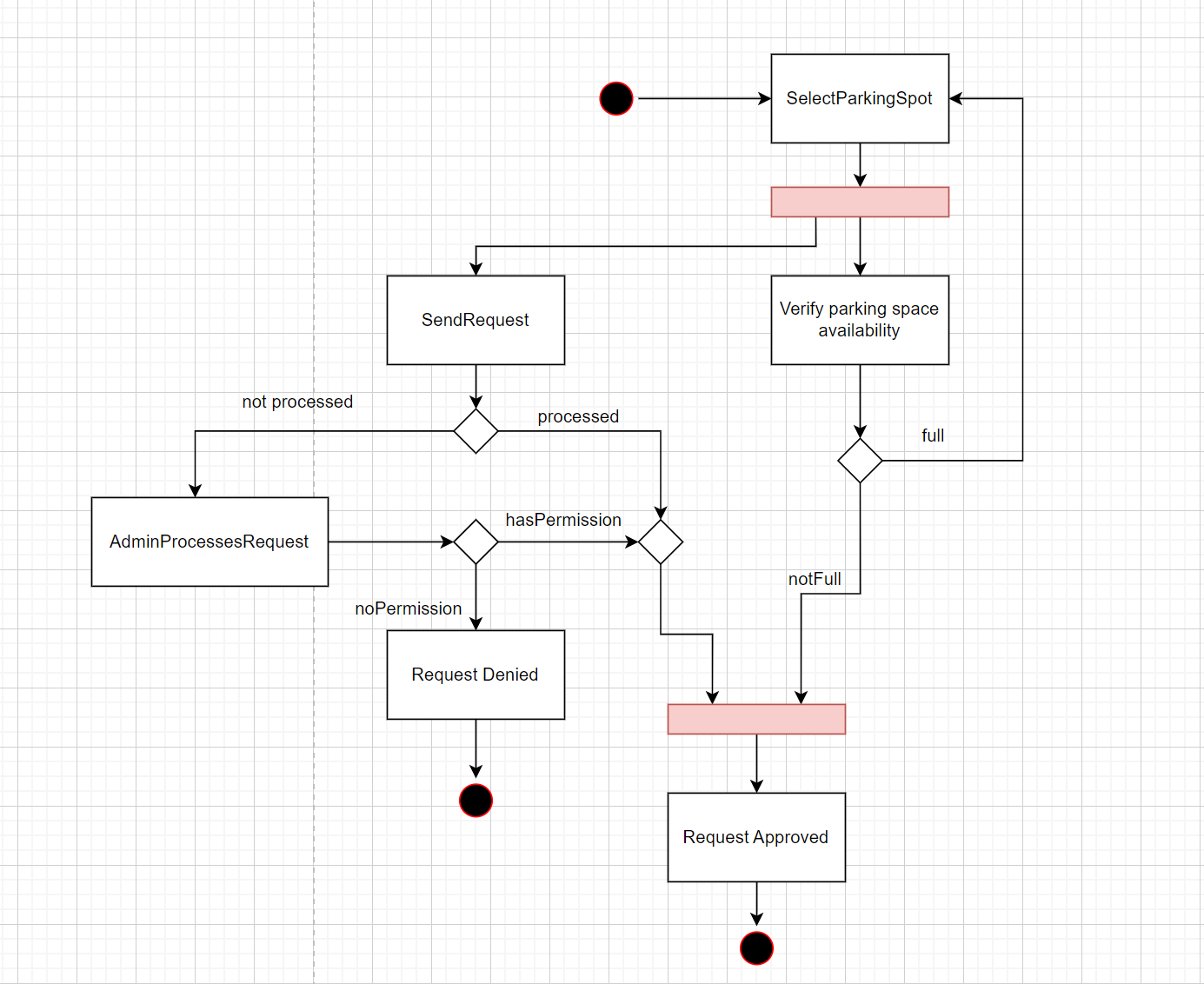
A screen shot of a computer program

Description automatically generated

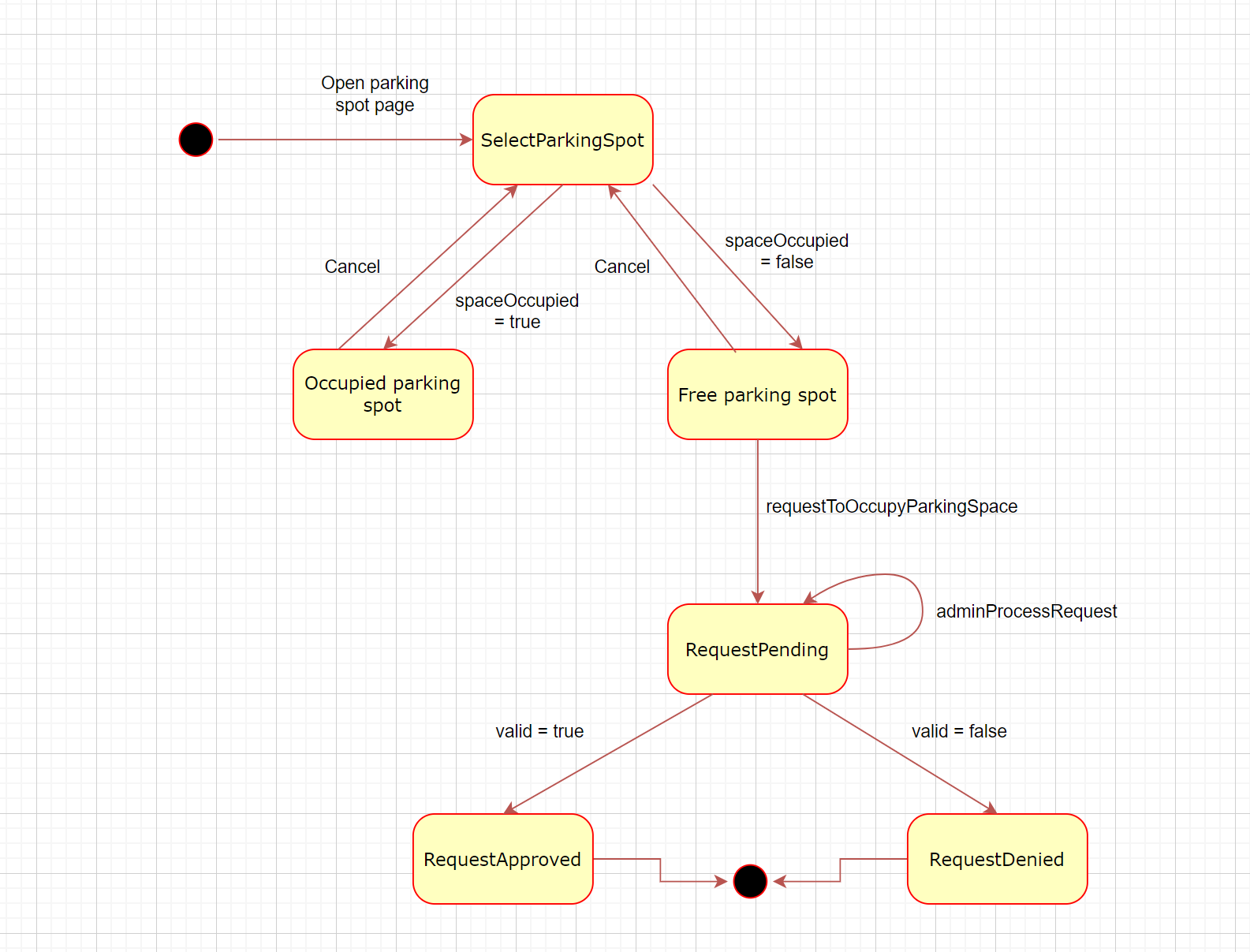
**Ricu Alexandru Razvan**

Eu am facut diagramele de state si activity pentru parking-reservation:

* **Activity diagram**



* **State diagram**



Pentru a gestiona fluxul de date și a menține o arhitectură clară, am ales pattern-ul Observer, implementat cu ajutorul bibliotecii Redux. Această decizie mi-a permis să:

* **Centralizez starea aplicației:** Toate datele partajate între componente (cum ar fi utilizatorul curent, utilizatorii conectati din chat) sunt stocate într-un singur loc, Redux Store.
* **Îmbunătățim scalabilitatea:** Pe măsură ce aplicația crește în complexitate, Redux ne permite să adăugăm noi funcționalități fără a afecta restul codului.
* **Optimiza performanța:** Prin mecanismul de subscribe, componentele sunt re-renderizate doar atunci când datele din store se modifică, ceea ce duce la o experiență mai fluidă pentru utilizator.
* **Simplifica managementul side effects:** Acțiuni precum apelurile către API-urile Firebase pot fi gestionate într-un mod predictibil și testabil, folosind middleware-uri Redux.
* În secțiunea de rezervări parking, Redux simplifică procesul de solicitare a locurilor de parcare, deoarece stochează datele utilizatorului curent. Astfel, majoritatea informațiilor necesare pentru apelurile API sunt preumplute automat, eliminând nevoia de a transmite aceste date manual prin componentele părinte.
* În cazul chat-ului, Redux stochează informații despre utilizatorii conectați, facilitând sincronizarea cu Firebase și asigurând o experiență de chat fluidă.
* Configuratia store-ul si ce date am in el:

****

A screen shot of a computer program

Description automatically generated

* În acest punct al codului, după ce utilizatorul a fost autentificat, se apelează funcția responsabilă cu salvarea datelor

A computer screen shot of a program code

Description automatically generated

* Toate informațiile sunt stocate într-un singur store global și pot fi accesate de orice componentă a aplicației

A screen shot of a computer program

Description automatically generated

* Un exemplu concret de cum Redux simplifică procesul de routing în aplicatie

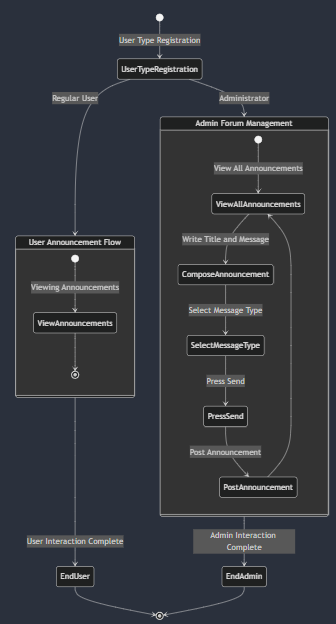
A screen shot of a computer program

Description automatically generated

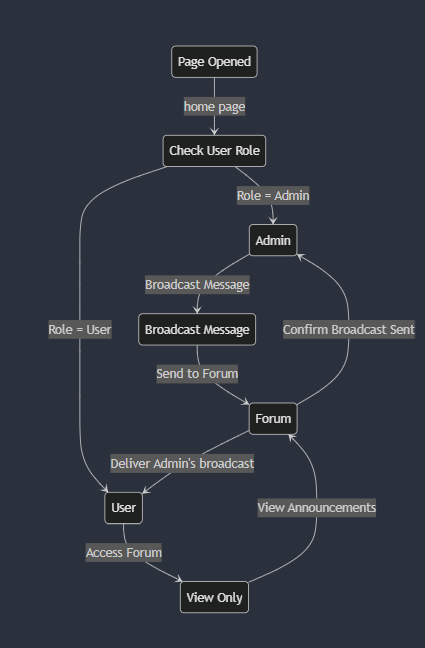
**Stan Mihai Alexandru**

Eu am facut diagrama pentru forumul din pagina de home:

1. Pentru functionalitea forumul am facut diagrama de activity

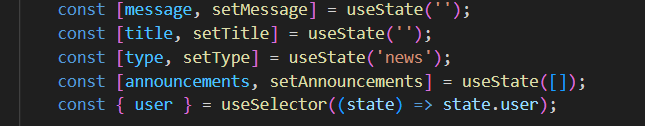


1. Pentru forum am realizat state diagram:

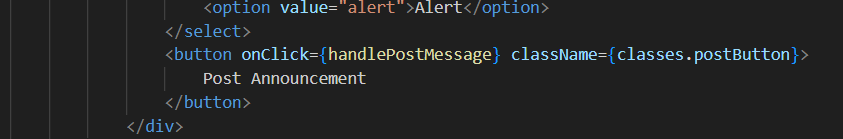


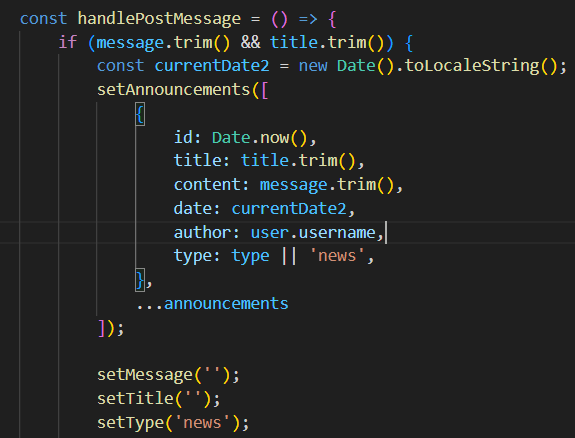
Pentru a gestiona schimbarea datelor in functie de inputului primit de la baza de date, precum si de la user inputs am ales sa folosesc state pattern. Avantajele acestuia sunt:

* **Claritate în gestionarea stărilor**: Comportamentele sunt organizate în stări clare, fiecare având o logică proprie.
* **Modularitate**: Permite adăugarea ușoară a noilor stări fără a modifica restul codului.
* **Reducerea complexității**: Împărțind logica pe stări distincte, codul devine mai ușor de înțeles și de întreținut.
* **Flexibilitate**: Permite schimbarea comportamentului unui obiect în mod dinamic, în funcție de starea acestuia.



* Atunci cand primesc un mesaj nou de la user, se modifica state-ul variabiler ce se ocupa de mesaje:





* Tot aceeasi variabila este modificata cand primesc la inceput data din baza de date



Am folosit aceleasi metode si la gestionarea datelor si variabilelor in cadrul masinii de spalat.

**Moga Eduard Mihai**

Diagramele facute de mine au fost 2 state diagrams si 1 activity diagram:

State diagram pentru functionalitate de register cu OCR

A diagram of a process

Description automatically generated

Activity diagram pentru functionalitate OCR

A diagram of a data flow

Description automatically generated

State diagram pentru OCRA diagram of a software system

Description automatically generated

Design pattern-ul ales de mine a fost **Observer** in contexul implementari unui chat app intre utilizatorii aplicatiei. Am ales acest design pattern deoarece dorim un mediu **event-driven** care sa declanseze evenimente si pentru restul participantiilor la conversatie.

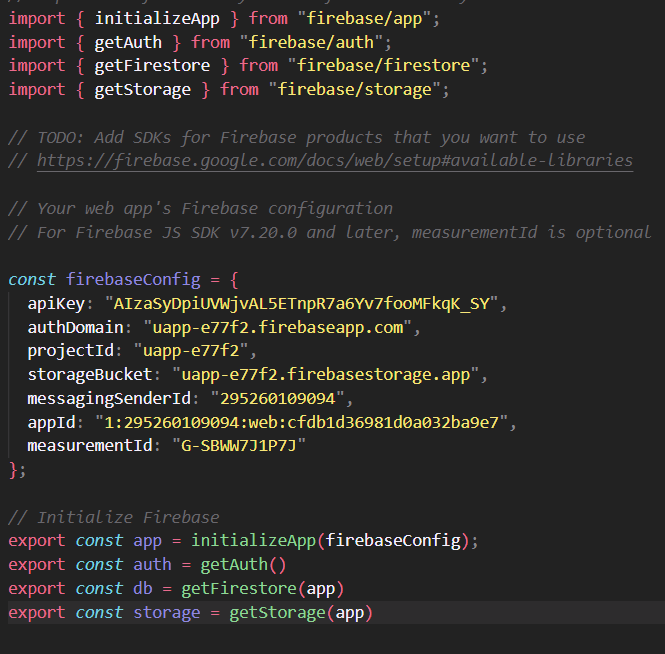
Astfel m-am folosit pentru stocarea de date si pentru implementarea chat-ului de Firebase cu Firestore si Firebase database.

Pentru a mentine state-urile in aplicatie am create chatSlice care contine detalii aferente pentru popularea documentelor(tabele) din cadrul Firebase database in stateuri diferite ale aplicatiei.

A screen shot of a computer program

Description automatically generated

Configurarea Firebase, pentru a folosi API necesare



La login se salveaza state-ul despre current user UID care ne permite in momently creari unui document sa avem aceasta valoare folosita la calcularea documentelor aferente: chat, chatUsers

A black screen with white text

Description automatically generated

A screen shot of a computer code

Description automatically generated In componenta de search, search.jsx facem cautarea unui user astfel aici vom avea nevoie de date corespunzatoare pentru a updata documentele cu updateDoc.

In componenta de input tratam functionalitatea de send a unor mesage implicand update-ul documentului char dar si a documentului userChats in care avem ca variabila ultimul mesaj trimis.

A computer screen shot of a program code

Description automatically generated

A screen shot of a computer program

Description automatically generated

Observerul joaca un rol important in urmatoarea functionalitate reactiva care ne ajuta sa avem o experienta seamless din punct de vedere al unui chat-app.

In componenta message, message.jsx avem un useEffect care are in lista de sensibilitati chatId, cand avem de lucru cu acesta se declanseaza useEffect-ul.

Apare si API call **onSnapshot()** care implementeaza Observer pattern. Astfel ca in momentul unui message setam state-ul mesajului din documentul chats.

A screen shot of a computer program

Description automatically generated