Case study solutions

Backend (Express.js + Node.js):
1. Voter and Candidate Routes:

```
// routes/voter.js
const express = require('express');
const router = express.Router();
const voterController = require('../controllers/voterController');

router.post('/register', voterController.register);
router.post('/login', voterController.login);
router.get('/history', voterController.getVotingHistory);

module.exports = router;

// routes/candidate.js
const express = require('express');
const router = express.Router();
const candidateController = require('../controllers/candidateController');

router.post('/add', candidateController.addCandidate);
router.put('/update/:id', candidateController.updateCandidate);
router.delete('/delete/:id', candidateController.deleteCandidate);

module.exports = router;
```

detailed controller functions for both the **voterController** and **candidateController** based on the routes you provided:
1. Voter Controller (controllers/voterController.js):

```
const voters = []; // This is a sample in-memory data store. In a real app, you'd use a database.

exports.register = (req, res) => {
  const voter = {
    id: Date.now(), // Simple unique ID. Use UUID or database auto-increment in real apps.
    ...req.body
  };
  voters.push(voter);
  res.status(201).json({
    message: 'Voter registered successfully',
    voter
  });
};
```

```
exports.login = (req, res) => {
  const { username, password } = req.body;
  const voter = voters.find(v => v.username === username && v.password === password);
  if (voter) {
    // In a real app, generate a JWT token or session here.
    res.json({
      message: 'Login successful',
      voter
    });
  } else {
    res.status(401).json({
      message: 'Invalid credentials'
    });
  }
};

exports.getVotingHistory = (req, res) => {
  // For simplicity, we're assuming the voting history is part of the voter's data.
  const voterId = req.params.id;
  const voter = voters.find(v => v.id == voterId);
  if (voter) {
    res.json(voter.votingHistory || []);
  } else {
    res.status(404).json({
      message: 'Voter not found'
    });
  }
};
```

2. Candidate Controller (controllers/candidateController.js):

```
const candidates = []; // This is a sample in-memory data store. In a real app, you'd use a database.

exports.addCandidate = (req, res) => {
  const candidate = {
    id: Date.now(), // Simple unique ID. Use UUID or database auto-increment in real apps.
    ...req.body
  };
  candidates.push(candidate);
  res.status(201).json({
    message: 'Candidate added successfully',
    candidate
  });
};

exports.updateCandidate = (req, res) => {
```

```
    const candidateId = req.params.id;
    const index = candidates.findIndex(c => c.id == candidateId);
    if (index !== -1) {
        const updatedCandidate = { ...candidates[index], ...req.body };
        candidates[index] = updatedCandidate;
        res.json({
            message: 'Candidate updated successfully',
            updatedCandidate
        });
    } else {
        res.status(404).json({
            message: 'Candidate not found'
        });
    }
};

exports.deleteCandidate = (req, res) => {
    const candidateId = req.params.id;
    const index = candidates.findIndex(c => c.id == candidateId);
    if (index !== -1) {
        candidates.splice(index, 1);
        res.json({
            message: 'Candidate deleted successfully'
        });
    } else {
        res.status(404).json({
            message: 'Candidate not found'
        });
    }
};
```

Installing Dependencies
You will need **express**, **body-parser**, and **nodemon** for auto-restarting the application during development:

```
npm install express body-parser
npm install nodemon --save-dev
```

3. Setting up Express Server (server.js)

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// Middleware to parse JSON data
```

```
app.use(bodyParser.json());

// Importing routes
const voterRoutes = require('./routes/voter');
const candidateRoutes = require('./routes/candidate');

// Using routes
app.use('/api/voters', voterRoutes);
app.use('/api/candidates', candidateRoutes);

// Setting up server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});

module.exports = app;
```

4. Voter Routes (routes/voter.js)

```
const express = require('express');
const router = express.Router();

// Sample data
let voters = [
  { id: 1, name: 'Voter 1' },
  { id: 2, name: 'Voter 2' }
];

// List all voters
router.get('/', (req, res) => {
  res.json(voters);
});

// Get a specific voter
router.get('/:id', (req, res) => {
  const voter = voters.find(v => v.id == req.params.id);
  res.json(voter);
});

// Add a new voter
router.post('/', (req, res) => {
  const newVoter = req.body;
  voters.push(newVoter);
  res.json(newVoter);
});
```

```
// Update a voter
router.put('/:id', (req, res) => {
   const updatedVoter = req.body;
   voters = voters.map(v => v.id == req.params.id ? updatedVoter : v);
   res.json(updatedVoter);
});

// Delete a voter
router.delete('/:id', (req, res) => {
   voters = voters.filter(v => v.id != req.params.id);
   res.json({ message: 'Voter deleted' });
});

module.exports = router;
```

5. Candidate Routes (routes/candidate.js)

```
const express = require('express');
const router = express.Router();

// Sample data
let candidates = [
   { id: 1, name: 'Candidate 1' },
   { id: 2, name: 'Candidate 2' }
];

// List all candidates
router.get('/', (req, res) => {
   res.json(candidates);
});

// Get a specific candidate
router.get('/:id', (req, res) => {
   const candidate = candidates.find(c => c.id == req.params.id);
   res.json(candidate);
});

// Add a new candidate
router.post('/', (req, res) => {
   const newCandidate = req.body;
   candidates.push(newCandidate);
   res.json(newCandidate);
});

// Update a candidate
router.put('/:id', (req, res) => {
   const updatedCandidate = req.body;
```

```
  candidates = candidates.map(c => c.id == req.params.id ? updatedCandidate : c);
  res.json(updatedCandidate);
});

// Delete a candidate
router.delete('/:id', (req, res) => {
  candidates = candidates.filter(c => c.id != req.params.id);
  res.json({ message: 'Candidate deleted' });
});

module.exports = router;
```

Start the Server
Add a **start** script to your **package.json**:

```
"scripts": {
  "start": "nodemon server.js"
}
```

Then run:

```
npm start
```

Frontend (Angular):

Let's start by integrating the **@ngrx/store** for state management. This will handle the various functionalities mentioned such as registering, voting, and accessing vote history.
For the sake of clarity, I'll provide an outline for the steps required:

1. **Setup**: Install the necessary NgRx packages.
2. **State**: Define the interface for your state.
3. **Action**: Define actions for the different functionalities.
4. **Reducer**: Handle state changes based on dispatched actions.
5. **Effects**: (Optional) Handle side effects, like communicating with an API.

1. Setup
Using your terminal, install the necessary NgRx packages:

```
npm install @ngrx/store @ngrx/effects
```

2. State
Define the main interface for the state:

```
// models/state.model.ts
export interface AppState {
  candidates: Candidate[];
  user: User;
  voteHistory: Vote[];
}
```

```
export interface Candidate {
   id: string;
   name: string;
   photoUrl: string;
   logoUrl: string;
   voteCount: number;
}

export interface User {
   id: string;
   name: string;
   hasVoted: boolean;
}

export interface Vote {
   userId: string;
   candidateId: string;
   timestamp: Date;
}
```

3. Actions

Define actions for functionalities:

```
// actions/app.actions.ts
import { createAction, props } from '@ngrx/store';

// User Registration
export const registerUser = createAction(
  '[User] Register',
  props<{ user: User }>()
);

// User Login
export const loginUser = createAction(
  '[User] Login',
  props<{ userId: string }>()
);

// Vote for Candidate
export const voteForCandidate = createAction(
  '[Vote] For Candidate',
  props<{ userId: string, candidateId: string }>()
);

// Get User Voting History
export const getUserVoteHistory = createAction(
  '[Vote] User History',
  props<{ userId: string }>()
```

```
);

// ... other actions as needed
```
4. Reducer
Define a reducer to handle changes:
```
// reducers/app.reducer.ts
import * as AppActions from '../actions/app.actions';
import { AppState } from '../models/state.model';

export const initialState: AppState = {
  // initial state values
};

export function appReducer(state: AppState = initialState, action: AppActions.ActionsUnion):
AppState {
  switch (action.type) {
    case AppActions.registerUser.type:
      // handle user registration
      return {...state, user: action.user};

    case AppActions.voteForCandidate.type:
      // handle voting logic, perhaps increment candidate voteCount
      return state;

    case AppActions.getUserVoteHistory.type:
      // handle fetching user vote history
      return state;

    // ... handle other actions

    default:
      return state;
  }
}
```
5. Effects (Optional)
For side effects, such as API communication, you can use NgRx effects. This is just an overview, and a real implementation would require setting up services and actual API calls:

```
// effects/app.effects.ts
import { Injectable } from '@angular/core';
import { Actions, ofType, createEffect } from '@ngrx/effects';
import { EMPTY } from 'rxjs';
import { catchError, mergeMap } from 'rxjs/operators';

import * as AppActions from '../actions/app.actions';
import { ApiService } from '../services/api.service';
```

```
@Injectable()
export class AppEffects {
  constructor(
    private actions$: Actions,
    private apiService: ApiService
  ) {}

  registerUser$ = createEffect(() => this.actions$.pipe(
    ofType(AppActions.registerUser),
    mergeMap(action => this.apiService.registerUser(action.user)
      .pipe(
        catchError(() => EMPTY)
      )
    )
  ));

  // ... similar effects for other actions
}

// effects/app.effects.ts
import { Injectable } from '@angular/core';
import { Actions, ofType, createEffect } from '@ngrx/effects';
import { EMPTY } from 'rxjs';
import { catchError, mergeMap } from 'rxjs/operators';

import * as AppActions from '../actions/app.actions';
import { ApiService } from '../services/api.service';

@Injectable()
export class AppEffects {
  constructor(
    private actions$: Actions,
    private apiService: ApiService
  ) {}

  registerUser$ = createEffect(() => this.actions$.pipe(
    ofType(AppActions.registerUser),
    mergeMap(action => this.apiService.registerUser(action.user)
      .pipe(
        catchError(() => EMPTY)
      )
    )
  ));

  // ... similar effects for other actions
}
```

1. Angular Service:

You'd have services that interact with your backend:

```
// services/voting.service.ts
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Store } from '@ngrx/store';
import { AppState } from '../store/app.state';
import * as VotingActions from '../store/voting/voting.actions';

@Injectable({
  providedIn: 'root'
})
export class VotingService {
  private API_URL = 'http://localhost:3000';

  constructor(private http: HttpClient, private store: Store<AppState>) {}

  login(voterID: string) {
    this.http.post(`${this.API_URL}/login`, { voterID }).subscribe(data => {
      this.store.dispatch(VotingActions.loginSuccess(data));
    });
  }

  vote(candidateID: string) {
    this.http.post(`${this.API_URL}/vote`, { candidateID }).subscribe(data => {
      this.store.dispatch(VotingActions.voteSuccess(data));
    });
  }

  // ... other methods like register, getHistory, etc.
}
```

2. NgRx Store:
Create actions, reducers, and selectors for voting:

```
// store/voting/voting.actions.ts
import { createAction, props } from '@ngrx/store';

export const loginSuccess = createAction('[Voting] Login Success', props<{ data: any }>());
export const voteSuccess = createAction('[Voting] Vote Success', props<{ data: any }>());
// ... other actions
```

```
// store/voting/voting.reducer.ts
import { createReducer, on } from '@ngrx/store';
import * as VotingActions from './voting.actions';

export const votingReducer = createReducer(
  { /* initial state */ },
  on(VotingActions.loginSuccess, (state, { data }) => {
    // Handle the data and return new state
  }),
  on(VotingActions.voteSuccess, (state, { data }) => {
    // Handle the data and return new state
  })
  // ... other handlers
);

// store/selectors/voting.selectors.ts
import { createSelector } from '@ngrx/store';

export const selectVotingData = (state) => state.voting;
// ... other selectors
```

. Manage Candidates

1. **Add Candidate** - To allow authorized personnel to add candidates.
2. **Edit Candidate** - Update candidate details.
3. **Delete Candidate** - To remove a candidate from the list.

2. View Voting Stats

1. **Get All Votes** - Retrieve all vote details.
2. **Get Candidate Votes** - Check votes for a specific candidate.

3. User Management

1. **Logout** - To log the user out of the system.

Let's create these actions:

```
// actions/app.actions.ts

// Candidate Management
export const addCandidate = createAction(
  '[Candidate] Add Candidate',
  props<{ candidate: Candidate }>()
);

export const editCandidate = createAction(
  '[Candidate] Edit Candidate',
  props<{ candidateId: string, changes: Partial<Candidate> }>()
);

export const deleteCandidate = createAction(
  '[Candidate] Delete Candidate',
```

```
  props<{ candidateId: string }>()
);

// Voting Stats
export const getAllVotes = createAction(
  '[Vote] Get All Votes'
);

export const getCandidateVotes = createAction(
  '[Vote] Get Candidate Votes',
  props<{ candidateId: string }>()
);

// User Management
export const logout = createAction(
  '[User] Logout'
);
```

Remember, for every action, you will usually want to define corresponding success and error actions to manage the side effects.
For instance:

```
// For adding a candidate
export const addCandidateSuccess = createAction(
  '[Candidate] Add Candidate Success',
  props<{ candidate: Candidate }>()
);

export const addCandidateError = createAction(
  '[Candidate] Add Candidate Error',
  props<{ error: any }>()
);

// ... Similarly, define success and error actions for other functionalities
```

Additionally, in your effects, you'll handle the asynchronous operations related to these actions. For example:

```
// effects/app.effects.ts

addCandidate$ = createEffect(() => this.actions$.pipe(
  ofType(AppActions.addCandidate),
  mergeMap(action => this.apiService.addCandidate(action.candidate)
    .pipe(
      map(candidate => AppActions.addCandidateSuccess({ candidate: candidate })),
      catchError(error => of(AppActions.addCandidateError({ error: error })))
    )
  )
));
```

```
// ... Add other effects for the corresponding actions
```

Selectors are functions that allow us to select a slice of the state from the store. Using NgRx, they can be composed together to construct new selectors based on existing ones. We'll create selectors based on our previous context:

1. Candidate Selectors:
To fetch details about the candidates, such as a specific candidate, all candidates, etc.

2. Voting Stats Selectors:
To retrieve data about votes and vote patterns.

3. User Management Selectors:
To get details about the currently logged-in user, voting history, etc.

Let's create some selectors:

```
// selectors/app.selectors.ts

import { createFeatureSelector, createSelector } from '@ngrx/store';
import { AppState } from '../reducers/app.reducer';

// Base App Selector
export const selectApp = createFeatureSelector<AppState>('app');

// 1. Candidate Selectors
export const selectAllCandidates = createSelector(
  selectApp,
  (state: AppState) => state.candidates
);

export const selectCandidate = (candidateId: string) => createSelector(
  selectAllCandidates,
  candidates => candidates.find(candidate => candidate.id === candidateId)
);

// 2. Voting Stats Selectors
export const selectVoteCountForCandidate = (candidateId: string) => createSelector(
  selectAllCandidates,
  candidates => {
    const candidate = candidates.find(c => c.id === candidateId);
    return candidate ? candidate.voteCount : 0;
  }
);

export const selectAllVotes = createSelector(
  selectApp,
  (state: AppState) => state.votes
);

// 3. User Management Selectors
```

```
export const selectCurrentUser = createSelector(
  selectApp,
  (state: AppState) => state.currentUser
);

export const selectVotingHistory = createSelector(
  selectCurrentUser,
  (currentUser) => currentUser ? currentUser.votingHistory : []
);

// ... Additional selectors can be added as needed
```

some commonly used features in applications like yours:

1. **Login Handler**
2. **Register Handler**
3. **Vote Handler**
4. **Vote History Handler**

NgRx structure.

1. Login Handler

We're assuming you're using a service (**ApiService**) to interact with your backend. So, for the **loginUser** action:

typescriptCopy code

```
// effects/app.effects.ts loginUser$ = createEffect(() => this.actions$.pipe(
ofType(AppActions.loginUser), mergeMap(action => this.apiService.loginUser(action.userId)
.pipe( map(user => AppActions.loginUserSuccess({ user: user })), catchError(error =>
of(AppActions.loginUserFailure({ error: error })))  ) ) ));
```

2. Register Handler

For the **registerUser** action:

```
// effects/app.effects.ts

registerUser$ = createEffect(() => this.actions$.pipe(
  ofType(AppActions.registerUser),
  mergeMap(action => this.apiService.registerUser(action.user)
    .pipe(
      map(user => AppActions.registerUserSuccess({ user: user })),
      catchError(error => of(AppActions.registerUserFailure({ error: error })))
    )
  )
));
```

3. Vote Handler

For the **voteForCandidate** action:

```
// effects/app.effects.ts

voteForCandidate$ = createEffect(() => this.actions$.pipe(
  ofType(AppActions.voteForCandidate),
```

```
  mergeMap(action => this.apiService.voteForCandidate(action.userId, action.candidateId)
    .pipe(
      map(vote => AppActions.voteForCandidateSuccess({ vote: vote })),
      catchError(error => of(AppActions.voteForCandidateFailure({ error: error })))
    )
  )
));
```

Vote History Handler
For the **getUserVoteHistory** action:

```
// effects/app.effects.ts

getUserVoteHistory$ = createEffect(() => this.actions$.pipe(
  ofType(AppActions.getUserVoteHistory),
  mergeMap(action => this.apiService.getUserVoteHistory(action.userId)
    .pipe(
      map(voteHistory => AppActions.getUserVoteHistorySuccess({ voteHistory: voteHistory
})),
      catchError(error => of(AppActions.getUserVoteHistoryFailure({ error: error })))
    )
  )
));
```

Also, remember to update the **Actions** file to include success and failure actions:

```
// actions/app.actions.ts

export const loginUserSuccess = createAction(
  '[User] Login Success',
  props<{ user: User }>()
);

export const loginUserFailure = createAction(
  '[User] Login Failure',
  props<{ error: any }>()
);

// ... similar success and failure actions for other functionalities
```

Angular Routing
Firstly, we'll need some additional components:

1. **RegisterComponent** - where voters can register.
2. **CandidateManagementComponent** - where admins can add, update or delete candidates.
3. **VoteHistoryComponent** - where voters can see their vote history.
4. **DashboardComponent** - a main dashboard where voters see an overview, perhaps current candidates.

Angular Components:
In your terminal, generate the components:
ng generate component register
ng generate component candidate-management
ng generate component vote-history
ng generate component dashboard

Angular Routing:
With the components in place, we can now extend our routing:

```
// app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { VotingComponent } from './voting/voting.component';
import { RegisterComponent } from './register/register.component';
import { CandidateManagementComponent } from './candidate-management/candidate-
management.component';
import { VoteHistoryComponent } from './vote-history/vote-history.component';
import { DashboardComponent } from './dashboard/dashboard.component';

const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' }, // Redirect to dashboard by default
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
  { path: 'vote', component: VotingComponent },
  { path: 'candidate-management', component: CandidateManagementComponent },
  { path: 'vote-history', component: VoteHistoryComponent },
  { path: 'dashboard', component: DashboardComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

AuthGuard for Admin Routes: It's recommended to secure the admin routes, like **candidate-management**. We can utilize an **AuthGuard** to do this:

```
// guards/auth.guard.ts
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree, Router } from
'@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private router: Router) {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean |
UrlTree> | boolean | UrlTree {

    // Example logic: check if the user is an admin.
    // This can be based on JWT token, local storage, or some state management logic.
    const isAdmin = false; // TODO: replace with actual logic

    if (!isAdmin) {
      // If not admin, redirect to login or dashboard
      this.router.navigate(['/login']);
      return false;
    }

    return true;
  }
}
```

Protect the **candidate-management** route:

```
{ path: 'candidate-management', component: CandidateManagementComponent,
canActivate: [AuthGuard] },
```

Remember to import and add the **AuthGuard** to the **providers** array in your app module. This is a more concrete routing structure. However, building a real-world application will require a combination of more features like forms, state management logic, and handling side-effects (like API calls with NgRx effects).

simplified relational database schema to support the Digital Dynamic Voting System (DDVS):

Tables:

1. **Candidates**
    - candidateID (Primary Key, Auto-increment)

- fullName (Text, Not Null)
- voteCount (Integer, Default 0)
- photoURL (Text)
- logoURL (Text)

2. **Voters**
    - voterID (Primary Key, Auto-increment)
    - uniqueVoterCode (Text, Not Null, Unique)
    - firstName (Text)
    - lastName (Text)
    - passwordHash (Text, Not Null) // Hashed password for security
    - votingHistoryID (Foreign Key to VotingHistory)
3. **VotingHistory**
    - historyID (Primary Key, Auto-increment)
    - voterID (Foreign Key to Voters)
    - candidateID (Foreign Key to Candidates)
    - votingTimestamp (DateTime, Default Current Timestamp)
4. **Admins**
    - adminID (Primary Key, Auto-increment)
    - username (Text, Not Null, Unique)
    - passwordHash (Text, Not Null) // Hashed password for security
5. **Logs**
    - logID (Primary Key, Auto-increment)
    - actionType (Text, Not Null) // E.g., "Vote cast", "Voter registered", etc.
    - actorID (Integer, Not Null) // Could be Voter or Admin ID
    - timestamp (DateTime, Default Current Timestamp)

For a NoSQL database like MongoDB, the structure can be different. You'd typically have collections instead of tables, and the relationship between data can be embedded or referenced.
For example, the VotingHistory can be embedded directly within the Voter's document, making the relationship more direct, and queries could potentially be more performant depending on the use case.
For the sake of brevity, here excluded some potential fields like email, address, etc., for voters, and additional metadata for candidates (like party affiliation, etc.). The schema can be extended and normalized based on specific requirements and the DB system in use.

the tables with some sample data using SQL, specifically for a relational database system like PostgreSQL:

```
-- Create Candidates table
CREATE TABLE Candidates (
    candidateID SERIAL PRIMARY KEY,
    fullName VARCHAR(255) NOT NULL,
    voteCount INT DEFAULT 0,
    photoURL VARCHAR(255),
    logoURL VARCHAR(255)
);
```

```
-- Sample data for Candidates table
INSERT INTO Candidates (fullName, photoURL, logoURL) VALUES
('John Doe', 'http://example.com/photos/johndoe.jpg',
'http://example.com/logos/johndoe.png'),
('Jane Smith', 'http://example.com/photos/janesmith.jpg',
'http://example.com/logos/janesmith.png');

-- Create Voters table
CREATE TABLE Voters (
   voterID SERIAL PRIMARY KEY,
   uniqueVoterCode VARCHAR(255) UNIQUE NOT NULL,
   firstName VARCHAR(255),
   lastName VARCHAR(255),
   passwordHash VARCHAR(255) NOT NULL
);

-- Sample data for Voters table
INSERT INTO Voters (uniqueVoterCode, firstName, lastName, passwordHash) VALUES
('A12345', 'Alice', 'Anderson', 'hashedpassword123'), -- Note: The password would be
hashed using a function in a real scenario.
('B67890', 'Bob', 'Brown', 'hashedpassword456');

-- Create VotingHistory table
CREATE TABLE VotingHistory (
   historyID SERIAL PRIMARY KEY,
   voterID INT REFERENCES Voters(voterID),
   candidateID INT REFERENCES Candidates(candidateID),
   votingTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Sample data for VotingHistory table
INSERT INTO VotingHistory (voterID, candidateID) VALUES
(1, 1),
(2, 2);

-- Create Admins table
CREATE TABLE Admins (
   adminID SERIAL PRIMARY KEY,
   username VARCHAR(255) UNIQUE NOT NULL,
   passwordHash VARCHAR(255) NOT NULL
);

-- Sample data for Admins table
INSERT INTO Admins (username, passwordHash) VALUES
('adminUser', 'hashedadminpass789'); -- Note: The password would be hashed using a
function in a real scenario.
```

```
-- Create Logs table
CREATE TABLE Logs (
   logID SERIAL PRIMARY KEY,
   actionType VARCHAR(255) NOT NULL,
   actorID INT NOT NULL,
   timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Sample data for Logs table
INSERT INTO Logs (actionType, actorID) VALUES
('Vote cast', 1),
('Voter registered', 2);
```

Please note:

1. The above SQL is structured for a PostgreSQL relational database.
2. The password hashes in the example are placeholder text. In a real-world scenario, you'd use a cryptographic hashing function to store passwords securely.
3. Adjustments might be needed based on the actual database environment you're using.

some sample data for you in table format for each of our defined tables:

1. **Candidates Table**

candidateID	fullName	voteCount	logoURL	photoURL
C001	John Doe	220	/logos/johndoe_logo.png	/photos/johndoe.png
C002	Jane Smith	180	/logos/janesmith_logo.png	/photos/janesmith.png
C003	Alice Johnson	255	/logos/alice_logo.png	/photos/alice.png

2. **Voters Table**

voterID	firstName	lastName	passwordHash	lastVoted
V001	Robert	Brown	$2a$12$somehashedpwexample1	2023-08-20 10:30:45
V002	Emily	Clark	$2a$12$somehashedpwexample2	2023-08-20 11:15:32
V003	William	Green	$2a$12$somehashedpwexample3	2023-08-20 12:05:20

3. **VotingHistory Table**

historyID	voterID	candidateID	votingTimestamp
H001	V001	C001	2023-08-20 10:30:45
H002	V002	C003	2023-08-20 11:15:32
H003	V003	C002	2023-08-20 12:05:20

4. **Admins Table**

adminID	username	passwordHash
A001	admin1	$2a$12$adminhashedpwexample1
A002	admin2	$2a$12$adminhashedpwexample2

5. **Logs Table**

logID	actionType	description	timestamp
L001	LOGIN	Voter V001 logged in	2023-08-20 10:30:00
L002	VOTE	Voter V001 voted for C001	2023-08-20 10:30:45
L003	LOGIN	Voter V002 logged in	2023-08-20 11:15:00

Note: These tables are merely mock data for visual representation. Password hashes are just placeholders. In a real scenario, hashes would be generated from the actual passwords, and you'd typically use cryptographic tools to produce them.