Creating a complete MERN stack application with login and registration functionalities involves a substantial amount of code, and it's not feasible to provide the entire code here due to its length. However, I can guide you through the step-by-step process and provide code snippets for each part of the application. Let's start by setting up the project structure and then build each component step by step.

Project Structure:

- `server` (Backend)
  - `models` (User model)
  - `routes` (API routes)
  - `config` (Secrets, configuration)
  - `index.js` (Server entry point)

- `client` (Frontend)
  - `src`
    - `components` (React components)
      - `Auth` (Registration and login components)
      - `Home` (Home component after login)
    - `api` (API calls)
    - `App.js` (React app entry point)

- `package.json` (Root package.json for backend)
- `client/package.json` (Package.json for frontend)

Backend Setup:

1. Set up the backend server using Express and MongoDB.
2. Create a `models` folder and define a `User` model to store user information.
3. Create routes in the `routes` folder for user registration and login.
4. Use `bcrypt` for password hashing and `jsonwebtoken` for token-based authentication.

Frontend Setup:

1. Set up the frontend using Create React App.
2. Create components for user registration and login.
3. Use React Router for navigation between components.
4. Implement API calls using Axios for communication with the backend.

Here's a brief overview of the code for different parts of the application:

1. **Backend (`server/index.js`):**

```javascript
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const bodyParser = require('body-parser');
const authRoutes = require('./routes/auth');

const app = express();

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

mongoose.connect('mongodb://localhost:27017/mern_blog_app', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  useCreateIndex: true,
});

app.use('/api', authRoutes);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

2. **Backend (`server/routes/auth.js`):**

```javascript
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');
const router = express.Router();

// Register user
router.post('/register', async (req, res) => {
  try {
    // ... (registration logic)
    // Create and sign JWT token
    const token = jwt.sign({ userId: user.id }, 'your_secret_key');
    res.json({ token });
  } catch (error) {
    res.status(500).json({ message: 'Registration failed' });
  }
});

// Login user
router.post('/login', async (req, res) => {
  try {
    // ... (login logic)
    // Create and sign JWT token
    const token = jwt.sign({ userId: user.id }, 'your_secret_key');
    res.json({ token });
  } catch (error) {
    res.status(500).json({ message: 'Login failed' });
  }
});

module.exports = router;
```

3. **Frontend (`client/src/components/Auth/Register.js`):**

```jsx
import React, { useState } from 'react';
import axios from 'axios';

const Register = () => {
  const [formData, setFormData] = useState({
    username: '',
    email: '',
    password: '',
  });

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await axios.post('/api/register', formData);
      localStorage.setItem('token', response.data.token);
      // Redirect to home or dashboard
    } catch (error) {
      console.error('Registration failed', error);
    }
  };

  return (
    <div>
      <h2>Register</h2>
      <form onSubmit={handleSubmit}>
        {/* Input fields */}
        <button type="submit">Register</button>
      </form>
    </div>
  );
};

export default Register;
```

4. **Frontend (`client/src/components/Auth/Login.js`):**

```jsx
import React, { useState } from 'react';
import axios from 'axios';

const Login = () => {
  const [formData, setFormData] = useState({
    email: '',
    password: '',
  });

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await axios.post('/api/login', formData);
      localStorage.setItem('token', response.data.token);
      // Redirect to home or dashboard
    } catch (error) {
      console.error('Login failed', error);
    }
  };

  return (
    <div>
      <h2>Login</h2>
      <form onSubmit={handleSubmit}>
        {/* Input fields */}
        <button type="submit">Login</button>
      </form>
    </div>
  );
};

export default Login;
```

5. **Frontend (`client/src/App.js`):**

```jsx
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Register from './components/Auth/Register';
import Login from './components/Auth/Login';

const App = () => {
  return (
    <Router>
      <div className="App">
        <Switch>
          <Route exact path="/register" component={Register} />
          <Route exact path="/login" component={Login} />
          {/* Add more routes */}
        </Switch>
      </div>
    </Router>
  );
};

export default App;
```

These code snippets provide a basic structure for setting up registration and login functionality in a MERN stack application. However, building a complete application involves many more considerations, such as user authentication with tokens, error handling, UI improvements, and connecting the frontend and backend. You'll need to expand upon these code snippets and implement the missing parts to create a functional and secure application.

//

Certainly! Here's an example of how you can define a `User` model using Mongoose in the `models` folder of your backend project:

Backend (`server/models/User.js`):

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

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    unique: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
});

const User = mongoose.model('User', userSchema);

module.exports = User;
```

In this example, the `User` model defines the structure of the user documents in the MongoDB collection. Each user has a `username`, `email`, `password`, and `createdAt` field. The `createdAt` field is automatically set to the current date and time when a new user is created.

To use this model, you would require it in your authentication routes or wherever you need to interact with the user data. For example, in your `routes/auth.js` file:

```javascript
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');
const router = express.Router();

// Register user
router.post('/register', async (req, res) => {
  try {
    const { username, email, password } = req.body;

    // Check if user already exists
    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(400).json({ message: 'User already exists' });
    }
    // Hash the password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    // Create a new user
    const newUser = new User({
      username,
      email,
      password: hashedPassword,
    });

    await newUser.save();

    // Create and sign JWT token
    const token = jwt.sign({ userId: newUser._id }, 'your_secret_key');

    res.json({ token });
  } catch (error) {
    res.status(500).json({ message: 'Registration failed' });
  }
});

// ... (other routes)

module.exports = router;
```

Remember to replace `'your_secret_key'` with your actual secret key for signing JWT tokens. This code snippet demonstrates how to create a new user using the `User` model, hash the password, and save the user data to the database.

Additionally, you should implement similar logic for the login route and other relevant parts of your application.