# User login code details

**Required Modules**

* User: A Mongoose model representing the user in the database.
* bcrypt: A library for hashing and comparing passwords.

**2. SignupController**

Handles user registration.

**Functionality:**

1. **Input Validation**:
   * Checks if all required fields (firstname, lastname, email, password, cpassword) are provided. Responds with a 422 status and an error message if not.
   * Ensures the password is at least 6 characters long.
   * Confirms that password and cpassword match.
2. **Duplicate User Check**:
   * Queries the database for an existing user with the provided email. If found, it responds with a 400 status and a "User already exists" message.
3. **User Creation**:
   * Creates a new user instance with the provided data.
   * Saves the user to the database.
   * Responds with a 201 status and a success message if successful.
4. **Error Handling**:
   * Logs any errors and responds with a 500 status and an "Internal Server Error" message.

**3. LoginController**

Handles user login.

**Functionality:**

1. **Input Validation**:
   * Ensures both email and password fields are provided. Responds with a 422 status if not.
2. **User Lookup**:
   * Searches for a user with the provided email. If no user is found, responds with a 400 status and an "Invalid Credentials" message.
3. **Password Comparison**:
   * Uses bcrypt.compare() to verify the provided password against the hashed password in the database.
   * Responds with a 400 status if the password does not match.
4. **Token Generation**:
   * Calls a generateAuthToken() method (defined in the User model) to generate a JWT token for the session.
   * Sets the token in a cookie (jwtoken) with a long expiration time.
5. **Success Response**:
   * Responds with a 200 status and a "Login Successful" message if everything checks out.
6. **Error Handling**:
   * Logs any errors and responds with a 500 status and an "Internal Server Error" message.

**4. ForgotController**

Handles password reset when the user forgets their password.

**Functionality:**

1. **Input Validation**:
   * Ensures all required fields (email, password, cpassword) are provided. Responds with a 422 status if not.
   * Checks that password and cpassword match.
2. **User Lookup**:
   * Searches for a user with the provided email. If no user is found, responds with a 400 status and an "Email not found" message.
3. **Password Update**:
   * Updates the user's password with the provided value.
   * Saves the updated user to the database.
4. **Success Response**:
   * Responds with a 200 status and a success message.
5. **Error Handling**:
   * Logs any errors and responds with a 500 status and an "Internal Server Error" message.

**5. LogoutController**

Handles user logout.

**Functionality:**

1. **Clear Cookie**:
   * Removes the jwtoken cookie from the client by using res.clearCookie().
2. **Success Response**:
   * Responds with a 200 status and a "Logout successful" message.
3. **Error Handling**:
   * Responds with a 500 status if an error occurs.

# Booking of the user:

**Required Modules**

* **Booking**: A Mongoose model representing the booking data in the database.

**2. Helper Function: getBookingList**

This function processes and transforms the booking data to create a detailed list of individual adventures.

**Input:**

* Takes an object {bookings }, where bookings is an array of booking documents.

**Functionality:**

1. **Iterate Over Bookings**:
   * For each booking in the bookings array, it processes the nested adventures field.
2. **Extract Adventures**:
   * Each adventure is stored under a parkName. The function:
     + Iterates over the parkName keys.
     + Extracts the list of adventureNames for each parkName.
3. **Flatten Data**:
   * Creates a flat structure for each adventure, including:
     + Booking details (address, email, date, etc.).
     + Park name.
     + Adventure name.
4. **Return**:
   * Returns a flat array of adventure details for all bookings.

**3. BookTicketController**

Handles booking ticket creation.

**Input:**

* Extracted from req.body:
  + userDetails: User-provided information (e.g., name, email, address, etc.).
  + selectedAdventures: Selected parks and adventures.

**Functionality:**

1. **Validation**:
   * Ensures that all required fields in userDetails (e.g., name, email, address, etc.) are filled.
   * Validates that at least one visitor (adult or children) is specified.
2. **Booking Number Generation**:
   * Counts total bookings in the database using Booking.countDocuments().
   * Generates a unique booking number by appending the count to a prefix (WWA10000).
3. **Data Preparation**:
   * Combines userDetails, the generated bookingnumber, and selectedAdventures into a new booking object.
4. **Database Operation**:
   * Creates a new Booking document and saves it to the database.
5. **Response**:
   * On success, responds with a 201 status and the generated booking number.
   * On failure, logs the error and responds with a 500 status.

**4. MyBookingController**

Fetches bookings for a specific email.

**Input:**

* Extracted from req.body:
  + email: The email address to search for bookings.

**Functionality:**

1. **Validation**:
   * Ensures the email field is provided.
   * If not, responds with a 422 status.
2. **Database Query**:
   * Searches for bookings with the provided email using Booking.find().
3. **Check Results**:
   * If no bookings are found, responds with a 400 status and an error message.
4. **Transform Data**:
   * Uses the getBookingList function to create a detailed list of adventures from the retrieved bookings.
5. **Response**:
   * On success, responds with a 200 status and the list of bookings.
   * On failure, logs the error and responds with a 500 status.