Backend Technical Analysis: Functions and Unit Tests

1. Backend Functions: Technical Analysis

1.1. src/city.js — City, Attraction, and Restaurant Management

Core Functions

- **readData / writeData**:
    - Read/write city data from/to data.json using synchronous filesystem operations.
    - Ensures all city/attraction/restaurant data is persisted between requests.
- **getUserCities(username)**:
    - Returns the last 10 cities for a user (or the latest if not logged in).
    - For each city, only the last 5 attractions and restaurants are included.
    - Implements per-user data isolation and enforces display limits.

Route Handlers

- **GET /**:
    - Returns the user's cities (or latest) using getUserCities.
    - No authentication required for viewing.
- **POST /**:
    - Adds or updates a city for the logged-in user.
    - Enforces authentication, city name presence, and a max of 10 cities per user.
    - Attractions/restaurants are limited to 5 each per city.
    - Updates the latest field for guest viewing.
- **DELETE /:cityName**:
    - Deletes a city for the logged-in user.
    - Enforces authentication and checks for city existence.
- **POST /:cityName/attractions**:
    - Adds an attraction to a city.
    - Enforces authentication, city existence, and a max of 5 attractions per city.
    - Prevents duplicates.
- **DELETE /:cityName/attractions/:attraction**:
    - Removes an attraction from a city.
    - Enforces authentication and city existence.
- **POST /:cityName/restaurants**:

    - Adds a restaurant to a city.
    - Enforces authentication, city existence, and a max of 5 restaurants per city.
    - Prevents duplicates.

- **DELETE /:cityName/restaurants/:restaurant**:

    - Removes a restaurant from a city.
    - Enforces authentication and city existence.

Technical Notes

- All data operations are synchronous for simplicity (not scalable for production).
- Data isolation is per user, with a latest field for guest access.
- All limits (cities, attractions, restaurants) are enforced both on creation and display.

1.2. src/user.js — User Authentication and Session Management

Core Functions

- **readUsers / writeUsers**:

    - Read/write user data from/to users.json using synchronous filesystem operations.

- **isValidEmail(email)**:

    - Validates email format using a simple regex.

Route Handlers

- **POST /register**:

    - Registers a new user if the email is valid and not taken.
    - Stores user in users.json and sets session.

- **POST /login**:

    - Logs in a user if credentials and email are valid.
    - Sets session on success.

- **POST /logout**:

    - Destroys the session to log out the user.

- **GET /me**:

    - Returns the current logged-in user's username, or 401 if not logged in.

Technical Notes

- All authentication is session-based (using express-session).
- Email validation is enforced for both registration and login.
- Passwords are stored in plaintext (not secure for production).

1.3. src/server.js — Main Server Setup

- Sets up Express, CORS (for frontend integration), JSON parsing, cookie parsing, and session management.
- Mounts user and city routers under /api/users and /api/cities.
- Provides a root health check endpoint.

---

2. Unit Tests: Deep Technical Analysis

2.1. tests/city.test.js — City, Attraction, and Restaurant Routes

Testing Approach

- Uses Vitest for test structure and assertions.
- Mocks Express request/response objects to directly invoke route handlers.
- Resets data.json before each test for isolation.
- Tests are synchronous and manipulate the filesystem directly.

Key Test Strategies

- **Mocking**:
    - Custom mockReqRes function creates mock req and res objects, simulating Express route calls without a running server.
    - Allows direct invocation of route logic for fast, isolated tests.
- **Data Reset**:
    - beforeEach writes an empty object to data.json to ensure no test data leaks between tests.
    - afterAll cleans up the file after all tests.

Test Coverage

- **CRUD Operations**:
    - Adding, deleting, and listing cities for users.
    - Adding attractions/restaurants to cities.
    - Removing attractions/restaurants.
- **Limit Enforcement**:
    - Tests for city limit (10 per user), attraction/restaurant limit (5 per city).
    - Verifies that exceeding limits returns correct error messages.
- **Data Isolation**:
    - Ensures users cannot access or modify each other's data.
- **Display Logic**:
    - Verifies that only the last 10 cities and last 5 attractions/restaurants are returned.
- **Error Handling**:
    - Tests for missing authentication, missing data, and invalid operations.

Example Test: Enforcing City Limit

```
it('enforces city limit (10 per user)', () => {
  const session = { user: { username: 'eve' } };
  // Add 10 cities
  for (let i = 0; i < 10; i++) {
    const city = { name: `City${i}`, attractions: [], restaurants: [] };
    const { req, res } = mockReqRes({ city }, session, {});
    cityRoutes.handle({ ...req, method: 'POST', url: '/' }, res, () => {});
    expect(res.statusCode).toBe(200);
  }
  // Try to add 11th city
  const city = { name: 'City10', attractions: [], restaurants: [] };
  const { req, res } = mockReqRes({ city }, session, {});
  cityRoutes.handle({ ...req, method: 'POST', url: '/' }, res, () => {});
  expect(res.statusCode).toBe(400);
  expect(res.jsonPayload).toHaveProperty('error');
});
```

- This test programmatically adds 10 cities, then attempts to add an 11th, expecting a 400 error and an error message.

2.2. tests/user.test.js — User Authentication and Session Management

Testing Approach

- Uses Vitest for structure and assertions.
- Mocks Express request/response objects for direct route handler invocation.
- Resets users.json before each test for isolation.

Key Test Strategies

- **Mocking**:
  - mockReqRes creates mock req and res objects for user routes.
- **Data Reset**:
  - beforeEach writes an empty array to users.json to ensure test isolation.

Test Coverage

- **Registration**:
  - Registers new users, checks for duplicate prevention, and validates email format.
- **Login**:
  - Logs in users with correct credentials, rejects invalid logins, and enforces email validation.
- **Persistence**:
  - Verifies that users are correctly written to and read from users.json.

- **Error Handling**:
    - Ensures appropriate error messages and status codes for invalid operations.

Example Test: Registration Fails with Invalid Email

```
it('registration fails with invalid email', () => {
  const { req, res } = mockReqRes({ username: 'notanemail', password: 'pw' }, {});
  userRoutes.handle({ ...req, method: 'POST', url: '/register' }, res, () => {});
  expect(res.statusCode).toBe(400);
  expect(res.jsonPayload).toHaveProperty('error');
  expect(res.jsonPayload.error).toMatch(/valid email/i);
});
```

- This test attempts to register with an invalid email and expects a 400 error and a specific error message.

2.3. tests/health.test.js — Health Check Endpoint

Testing Approach

- Uses Vitest and Supertest to spin up a minimal Express app with only the health endpoint.
- Sends a GET request to / and asserts the response status and message.

Test Coverage

- **Health Check**:
    - Confirms that the backend root endpoint responds with the expected message and status code, verifying server availability.

---

3. Technical Testing Best Practices Observed

- **Test Isolation**:
    - All tests reset their respective data files before each test, ensuring no state leakage and reliable, repeatable results.
- **Direct Route Invocation**:
    - By mocking requests and responses, tests can directly invoke route logic, making them fast and independent of network or server state.
- **Comprehensive Error Checking**:
    - Tests assert not only on success but also on all relevant error conditions, including limit enforcement, authentication, and data validation.
- **Coverage of Edge Cases**:
    - Tests cover not just the happy path but also edge cases like exceeding limits, invalid input, and data isolation between users.

---

4. Summary

- The backend is structured for clarity and testability, with all business logic encapsulated in route handlers and utility functions.
- Unit tests are thorough, isolated, and directly exercise the backend logic, ensuring robust enforcement of all business rules and error conditions.
- The use of synchronous file operations and in-memory session management is suitable for demo and test environments, but would need to be replaced for production scalability and security.