Backend Testing

Outline

The Brickwyze backend was tested for reliability and correctness using Jest. A total of 39 tests were conducted across 8 test suites. Unit tests, database tests and integration tests were all performed. All tests conducted focused on key areas including:

- **Score Calculation:** Ensured that the core weighted scoring algorithm that combines foot traffic, demographics, crime, flood risk, rent and points of interest (POI) with custom weights to generate resilience scores.
- **Filter Logic:** Tested the demographic filtering functions including age bracket matching, ethnicity percentage calculations, and rent-zone based filtering.
- **Data Validation & Safety:** Ensured that null values, division by zero protection, and data validation were handled in a robust manner. This is so that system failures do not occur.
- **Database Structure Integrity:** Validated table structures, relationships, and data constraints across all tables.
- **API Request Handling:** Tested basic API endpoint as well as request/response validation.

List of Tests and Expected Outcomes

Test Category	Test Suite	Test Name	Description	Expected Outcome
Database	test-data-validation.ts	GEOID Validation	Verifies GEOID follows 11-digit NYC format	Valid GEOIDs match regex pattern, invalid ones are rejected
Database	test-data-validation.ts	Should ensure GEOID uniqueness	Tests that no duplicate GEOIDs exist in the dataset	Set size equals array length, confirming uniqueness
Database	test-data-validation.ts	Score Validation	Confirms resilience scores are within 0-1 bounds	Valid scores (0.75) pass, invalid scores (-0.1, 1.5) fail
Database	test-data-validation.ts	It should validate component scores 0-10	Tests that foot traffic, crime, POI scores are 0-10	Valid scores (0, 5.5, 7.2, 10) pass
Database	test-data-validation.ts	Percentage Validation	Ensure demographics percentages stay within the valid range	Valid percentages (0, 48.5, 100) pass and invalid percentages (-5, 150) fail.
Database	test-data-validation.ts	Should validate gender percentages add to 100	Tests male and female percentages total to approximately 100%	Total percentage = 100% (male is 48.5% and female is 51.5%)

Database	test-business-logic.ts	should reflect NYC income diversity	Validates representation across low, middle, high income brackets	All income levels (45, 120, 25) are greater than 0
Database	test-business-logic.ts	should reflect NYC ethnic diversity	Tests multi-ethnic representation in NYC demographics	All ethnicities (WEur: 450, BAfrAm: 380, HMex: 275, AEA: 200) > 0
Database	test-business-logic.ts	should calculate data completeness	Computes percentage of complete vs null records	2 complete records out of 3 total = 67% completeness
Database	test-table-structure.ts	should have required zone properties	Validates resilience zones table has essential fields	Zone object contains GEOID, resilience_score, foot_traffic_score, avg_rent
Database	test-table-structure.ts	should have required demographic properties	Tests demographics table contains population and age data	Demo object has GEOID, Total population, Male/Female %, Median age
Database	test-table-structure.ts	should have required income properties	Verifies economics table has income bracket data	Economics object contains GEOID, income brackets (HHIU10E, HHI50t74E), median income
Database	test-table-structure.ts	should have required ethnicity properties	Tests ethnicity table structure and population fields	Ethnicity object has GEOID, total_population, WEur, BAfrAm fields
Database	test-table-structure.ts	should have required crime properties	Validates crime trends table has historical/predicted data	Crime object contains GEOID, year_2024, pred_2025 fields
Database	test-table-structure.ts	should link tables by GEOID	Tests foreign key relationships between tables	Zone GEOID matches demographic GEOID for proper joins
Unit	test-filter-logic.ts	should calculate age percentages for working age	Tests age bracket filtering for 25-29 year olds	Returns 0.15 (15%) for working age demographic in test data
Unit	test-filter-logic.ts	should calculate age percentages for seniors	Tests age bracket filtering for 65+ population	Returns 0.08 (8%) for senior demographic in test data

Unit	test-filter-logic.ts	should calculate ethnicity percentages for single ethnicity	Tests filtering for single ethnic group (WEur)	Returns 0.6 (60% of 1000 population = 600 WEur residents)
Unit	test-filter-logic.ts	should calculate ethnicity percentages for multiple ethnicities	Tests combined ethnic group filtering (WEur + BAfrAm)	Returns 0.9 (90% = 600 WEur + 300 BAfrAm out of 1000)
Unit	test-filter-logic.ts	should filter zones by rent range	Tests rent affordability filtering with $1000-2000 range	Returns 3 zones: one in range, watched zone, null rent zone
Unit	test-filter-logic.ts	should include zones with null rent	Verifies zones with missing rent data pass through filters	Zones with null avg_rent are included in all rent filter results
Unit	test-filter-logic.ts	should always include watched zones	Tests watched zones override rent filtering restrictions	Watched zone (36061019500) appears regardless of $4000 rent vs $0-100 filter
Unit	test-filter-logic.ts	should handle empty filter arrays	Tests system behavior with no ethnicity criteria specified	Empty ethnicity array returns 0% match rate as expected
Unit	test-data-validation.ts	should handle safe division with zero denominator	Tests division by zero protection in calculations	Returns 0 when dividing by 0, prevents system crashes
Unit	test-data-validation.ts	should handle safe property access with null values	Tests property access with null/undefined values	Returns 0 for missing properties, maintains system stability
Unit	test-data-validation.ts	should find maximum percentage correctly	Tests normalization maximum detection for scoring	Finds max value 2.3, applies 1.0 minimum default for normalization
Unit	test-data-validation.ts	should handle realistic edge function data	Tests validation with incomplete real-world zone data	Handles null foot_traffic_score, undefined crime_score easily
Unit	test-score-calculation.ts	should calculate score with default weights	Tests weighted scoring with standard weight distribution	Returns 0.69 for sample inputs with default weights (35% foot traffic, 25%

				demographic, etc.)
Unit	test-score-calculation.ts	should handle null demographic score	Tests scoring when demographic data is unavailable	Returns 0.54 using 0 fallback for null demographic component
Unit	test-score-calculation.ts	should apply custom weights correctly	Tests user-defined weight preferences vs defaults	Custom weights (60% foot traffic, 20% demographic) produce 0.744 score
Unit	test-score-calculation.ts	should use default weights when no custom weights provided	Verifies default weight object returned for empty input	Returns standard weight distribution object unchanged
Unit	test-score-calculation.ts	should apply single custom weight	Tests partial customization while preserving other defaults	foot_traffic changes to 50%, demographic stays 25% (unchanged)
Unit	test-score-calculation.ts	should handle perfect scores	Tests scoring with maximum input values (1.0)	Perfect input across all factors returns 1.0 final score
Unit	test-score-calculation.ts	should handle zero scores	Tests scoring with minimum input values (0.0)	Zero input across all factors returns 0.0 final score
Unit	test-score-calculation.ts	should verify weights sum to 1.0	Mathematical validation that weights are properly normalized	Sum of all default weights equals 1.0 within precision tolerance
Integration	test-filter-logic.ts	should validate filter parameters	Tests API filter parameter structure validation	Confirms arrays and ranges have proper format and types
Integration	test-filter-logic.ts	should handle empty filters	Tests API behavior with empty filter objects	System accepts empty filter object without errors
Integration	test-basic-requests.ts	should make basic request and return zones	Tests basic API response structure and content	Returns zones array with geoid, resilience_score, custom_score properties
Integration	test-basic-requests.ts	should validate response structure	Tests API response data validation and formatting	Validates GEOID format, score ranges, and proper JSON structure

Test Success/Fail Rate

All tests passed successfully, confirming the backend system is functioning as expected under a variety of scenarios. Testing conducted is comprehensive handling edge cases, data validation as well as mathematical operations.

Test Category	Description	Total Tests	Passed	Failed	Success Rate
Score Calculation	Core weighted scoring algorithm to generate custom resilience scores	8	8	0	100%
Filter Logic	Demographic filtering function	8	8	0	100%
Data Validation & Safety	Handling of null values, division by zero protection and data validation to present system failures	10	10	0	100%
Database Structure Integrity	Validation of table structures, relationships and database constraints across all tables	9	9	0	100%
API Request Handling	Basic API endpoint testing and request/respons e validation	2	2	0	100%
Total		**39**	**39**	**0**	**100%**