Final Backend Testing

Outline

The backend of Brickwyze was tested for reliability and correctness using Jest. Following the development of new frontend features, the backend logic was refined, extended, and consequently retested vigorously. This resulted in the splitting of the logic into six modular components. A total of 177 tests (55 database tests, 97 unit tests and 25 integration tests) were conducted across 12 test suites. All tests conducted were focused on key areas including:

- **Score Calculation:** Ensured that the core weighted scoring algorithm that combines foot traffic.
- **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-business-logic.ts	should validate NYC ethnic diversity patterns	Tests fetchAllData with mockEthnicData containing WEur, BAfrAm, HMex, AEAKrn, AEA values and validates business rules for ethnic representation	All ethnic groups > 0 (WEur, BAfrAm, HMex, AEAKrn, AEA), all percentages < 80
Database	test-business-logic.ts	should validate NYC income diversity distribution	Tests fetchAllData with mockIncomeData containing low/middle/high income brackets and validates income distribution rules	result.incomeData.length = 3, all counts > 0, middle income count > low and high income counts
Database	test-business-logic.ts	should validate resilience score calculation weights	Tests fetchAllData with mockZoneScores containing individual score components and validates weight calculation	footTrafficContribution > crimeContribution and floodContribution, totalWeight = 0.75
Database	test-business-logic.ts	should validate score ranges and constraints	Tests fetchAllData with mockScoreValidation	min_resilience ≥ 0, max_resilience ≤ 10, validPercentage > 95%

			containing min/max resilience and zone count data	
Database	test-business-logic.ts	should validate top zone identification logic	Tests processZones with mockTopZones containing ranked zones and validates ranking logic	result[0].custom_score > result[1].custom_score, ranks = 1 and 2, all zones meet rent (3000-4000) and korean_percent > 10 criteria
Database	test-business-logic.ts	should validate zone filtering by demographic preferences	Tests calculateDemographicPercentages with mockInput and validates demographic filtering	result.ethnicPercent['36061019500'] ≥ 0.20, result.genderPercent['36061019500'] ≥ 0.50
Database	test-business-logic.ts	should validate minimum data completeness requirements	Tests fetchAllData with mockCompletenessData (100 zones, 98 ethnicity, 95 demographics, 92 income) and validates completeness	ethnicityCompleteness > 85%, demoCompleteness > 85%, isValid = true
Database	test-business-logic.ts	should calculate data completeness score for individual zones	Tests completeness calculation with zoneData array containing 3 complete and 2 null records	completenessPercentage = 60%, completenessPercentage ≥ 60
Database	test-business-logic.ts	should validate Manhattan GEOID patterns	Tests GEOID format validation	Manhattan GEOIDs (those beginning with 36061) return true, others return false
Database	test-business-logic.ts	should validate resilience score distribution by borough	Tests fetchAllData with mockBoroughScores for Manhattan, Brooklyn, Queens and validates borough scoring rules	All zones have resilience_score between 4.0-8.0, Manhattan score > Brooklyn score
Database	test-complex-queries.ts	should join all tables for comprehensive zone data	Tests fetchAllData with mockCompleteData containing all zone attributes (resilience_score, avg_rent, crime data, traffic data, demographics, ethnicity, income)	fetchAllData called with mockSupabase, result.zones[0].resilience_score = 6.2, result.ethnicityData[0].korean_percent = 12.5, result.incomeData[0].median_income = 85000
Database	test-complex-queries.ts	should join crime and foot traffic trends for time series analysis	Tests fetchCrimeData and fetchFootTrafficData with mockTrendData containing temporal	crimeResult[0].crime_pred_2025 = 0.25, trafficResult[0].traffic_pred_2025 = 0.9

			data (crime_2024, crime_pred_2025, traffic data by period)	
Database	test-complex-queries.ts	should filter zones by multiple demographic criteria	Tests processZones with mockFilteredData and mockInput containing rentRange, ethnicities, incomeRange, ageRange filters	processZones called, result.length = 2, result[0].resilience_score > result[1].resilience_score
Database	test-complex-queries.ts	should filter zones by income distribution patterns	Tests fetchAllData with mockIncomeFilteredData containing income diversity metrics	result.incomeData[0].income_diversity_score > 0.5
Database	test-complex-queries.ts	should calculate zone statistics by borough	Tests fetchAllData with mockBoroughStats containing Manhattan (GEOID: '36061019500') and Brooklyn (GEOID: '36047019500') data	result.zones.length = 2, result.zones[0].resilience_score > result.zones[1].resilience_score
Database	test-complex-queries.ts	should find top zones by combined criteria	Tests processZones with mockTopZones containing two zones with different custom_score values and mockInput with rent/ethnicity filters	result[0].custom_score > result[1].custom_score
Database	test-complex-queries.ts	should check data completeness across all tables	Tests fetchAllData with mockCompletenessData (100 zones, 97 ethnicity, 98 demographics, 95 income) and validateDatabaseData	completenessPercentage > 85%, result.zones.length > 0, isValid = true
Database	test-complex-queries.ts	should identify zones with missing critical data	Tests fetchAllData with mockMissingDataZones having 3 zones but missing ethnicity/demographics data for some zones, then analyzes missing data patterns	missingEthnicity.length > 0, missingDemographics.length > 0
Database	test-complex-queries.ts	should test complete pipeline with demographic scoring	Tests complete data pipeline: fetchAllData with mockZones and mockEthnicityData, then calcDemo with	data.zones.length = 1, percentages.ethnicPercent['36061019500'] = 0.152

			mockInput for demographic percentage calculation	
Database	test-complex-queries.ts	should test zone processing with all filters	Tests processZones with mockProcessedZones and mockInput containing multiple weight filters and criteria	result[0].custom_score > 80, result[0].demographic_score > 70
Database	test-constraints-and-validations.ts	should validate GEOID format for NYC census tracts	Tests GEOID format validation for NYC census tracts - validates 11-digit format starting with 36061	Valid GEOID matches /^\d{11}$/ and starts with '36061', invalid GEOID fails validation
Database	test-constraints-and-validations.ts	should ensure GEOID uniqueness in resilience_zones	Tests for duplicate GEOID scenarios in resilience zones data	Duplicate GEOIDs detected when length of original array > unique set length
Database	test-constraints-and-validations.ts	should enforce foot_traffic_score constraints	Tests foot traffic score normalization and clamping to valid 0-10 range	Invalid scores (15.0, -1.0, 150, -50) get clamped to 0-10 range, valid scores unchanged
Database	test-constraints-and-validations.ts	should enforce crime_score constraints	Tests crime score clamping to valid 0-10 range	Score of 11.5 gets clamped to ≤10 and ≥0
Database	test-constraints-and-validations.ts	should accept valid score ranges	Tests that valid scores (0, 5.5, 7.2, 10) are accepted	All valid scores are ≥0 and ≤10
Database	test-constraints-and-validations.ts	should validate request body constraints	Tests validation of request body parameters including weights, ethnicities, rentRange, and topN	Valid request passes validation, invalid requests with weight values >100, negative rent, or topN ≤0 throw errors
Database	test-constraints-and-validations.ts	should validate database data completeness	Tests database data completeness validation	Valid data with zones returns true, empty zones data returns false
Database	test-constraints-and-validations.ts	should validate percentage ranges (0-100)	Tests percentage validation for values between 0-100	Valid percentages (0, 48.5, 51.5, 100) pass validation, invalid percentages (-5, 150) fail
Database	test-constraints-and-validations.ts	should validate gender percentages sum to 100	Tests that male and female percentages sum to 100	Male percent (48.5) + Female percent (51.5) = 100.0
Database	test-constraints-and-validations.ts	should handle null values appropriately	Tests handling of null values in database data	Null values are preserved and handled correctly (avg_rent = null)
Database	test-constraints-and-validations.ts	should validate automatic	Tests automatic resilience score	Calculated score matches expected formula: (foot_traffic * 0.35) + (crime

		resilience score calculation	calculation using weighted formula	* 0.15) + (flood_risk * 0.10) + (rent * 0.10) + (poi * 0.05)
Database	test-constraints-and-validations.ts	should validate score normalization	Tests score normalization to 0-100 range	Scores [-10, 0, 50, 100, 150] normalize to [0, 0, 50, 100, 100]
Database	test-constraints-and-validations.ts	should validate population consistency across tables	Tests population consistency between ethnicity and demographics tables	Population difference between tables is <5%
Database	test-constraints-and-validations.ts	should validate missing data handling	Tests handling of completely missing data scenarios	Empty data arrays return length 0
Database	test-constraints-and-validations.ts	should validate GEOID consistency across tables	Tests GEOID consistency across zones, ethnicity, demographics, and income tables	Missing GEOIDs identified: 1 missing from ethnicity, 1 missing from income
Database	test-constraints-and-validations.ts	should validate borough name resolution	Tests borough name resolution from GEOID county codes	Correct borough mapping: “061”: Manhattan, “005”: Bronx, “047” : Brooklyn, “081”: Queens, “085”: Staten Island, “999”: Unknown
Database	test-crud-operations.ts	should fetch resilience zone data	Tests fetchAllData with mockZoneData containing sample zone data and validates data retrieval	fetchAllData called with mockSupabase, result.zones.length = 1, result.zones[0].GEOID = '36061019500', result.zones[0].avg_rent = 3500.00
Database	test-crud-operations.ts	should handle zone data updates through processZones	Tests processZones with updated zone data containing custom scores and validates zone processing	processZones called, result[0].custom_score = 8.5, result[0].avg_rent = 3800
Database	test-crud-operations.ts	should filter zones by rent range	Tests fetchAllData with multiple zones and validates rent-based filtering logic	filteredZones.length = 2, filtered GEOIDs = ['36061019500', '36061019700'] (rent range 3000-4000)
Database	test-crud-operations.ts	should validate zone data integrity	Tests validateDatabaseData with valid and invalid zone data	Valid data with zones returns true, empty zones data returns false
Database	test-crud-operations.ts	should fetch crime trend data	Tests fetchCrimeData with mock crime data and validates crime data retrieval	fetchCrimeData called with mockSupabase and GEOIDs, result.length = 1, result[0].GEOID = '36061019500', result[0].year_2024 = 0.3

Database	test-crud-operations.ts	should handle crime data updates via addCrimeDataToTopZones	Tests addCrimeDataToTopZones with mock top zones and validates crime data enrichment	addCrimeDataToTopZones called, mockTopZones[0].crime_timeline defined, mockTopZones[0].crime_trend_direction = 'decreasing'
Database	test-crud-operations.ts	should handle missing crime data gracefully	Tests fetchCrimeData with invalid GEOID and validates null data handling	result = null for invalid GEOID
Database	test-crud-operations.ts	should fetch foot traffic data	Tests fetchFootTrafficData with mock traffic data and validates data retrieval	fetchFootTrafficData called, result.length = 1, result[0].morning_2024 = 0.8, result[0].afternoon_2024 = 0.7
Database	test-crud-operations.ts	should handle foot traffic data enrichment	Tests addFootTrafficDataToTopZones with mock zones and validates traffic data enrichment	addFootTrafficDataToTopZones called, mockTopZones[0] has foot_traffic_timeline, foot_traffic_periods_used, and foot_traffic_by_period properties
Database	test-crud-operations.ts	should retrieve foot traffic by time period	Tests fetchFootTrafficData with time period data and validates period-specific retrieval	result[0].morning_2024 = 0.8, result[0].afternoon_2024 = 0.7, result[0].evening_2024 = 0.6
Database	test-crud-operations.ts	should fetch demographic data	Tests fetchAllData with mock demographic data and validates demographic data retrieval	result.demographicsData.length = 1, result.demographicsData[0]['Total population'] = 5000, Male % = 48.5, Female % = 51.5
Database	test-crud-operations.ts	should validate demographic data consistency	Tests demographic data validation including gender percentages and population values	Gender sum ≈ 100%, population > 0, median age > 0 and < 120
Database	test-crud-operations.ts	should fetch race/ethnicity data	Tests fetchAllData with mock ethnicity data and validates ethnicity data retrieval	result.ethnicityData.length = 1, result.ethnicityData[0].AEAKrn = 12.5, result.ethnicityData[0].WEur = 35.2, total_population = 5000
Database	test-crud-operations.ts	should validate ethnicity percentages are reasonable	Tests ethnicity data validation including total ethnic percentages and non-negative values	Total ethnic ≤ total population, all ethnicity values ≥ 0

Database	test-crud-operations.ts	should validate ethnicity percentages are reasonable	Tests ethnicity data validation including total ethnic percentages and non-negative values	Total ethnic ≤ total population, all ethnicity values ≥ 0
Database	test-crud-operations.ts	should validate economic data ranges	Tests economic data validation including median income and household count ranges	Median income > 0 and < 1000000, household counts ≥ 0
Database	test-crud-operations.ts	should handle bulk zone data fetching	Tests fetchAllData with multiple zones and validates bulk data retrieval	result.zones.length = 3, GEOIDs = ['36061019500', '36061019600', '36061019700']
Database	test-crud-operations.ts	should handle bulk zone processing	Tests processZones with multiple input zones and validates bulk processing	result.length = 2, result[0].custom_score > result[1].custom_score
Database	test-crud-operations.ts	should handle missing data in bulk operations	Tests fetchAllData with incomplete data and validates missing data handling	result.zones.length = 2, result.ethnicityData.length = 1, missingEthnicityData contains '36061019600'
Unit	test-data-processing.ts	should fetch all data successfully	Tests fetchAllData with mockSupabase and validates successful data retrieval from all tables	result.zones defined, result.ethnicityData defined, result.demographicsData defined, result.incomeData defined, mockFetchAllData called with mockSupabase
Unit	test-data-processing.ts	should throw error when zones fetch fails	Tests fetchAllData error handling when zones fetch fails with connection error	Throws error with 'Failed to fetch zones' message
Unit	test-data-processing.ts	should handle zones error response	Tests fetchAllData error handling when zones fetch fails with database error	Throws error with 'Failed to fetch zones' message
Unit	test-data-processing.ts	should handle optional data failures gracefully	Tests fetchAllData with zones data present but optional data (ethnicity, demographics) null or empty	result.zones defined, result.ethnicityData = null, result.demographicsData = null, result.incomeData = []
Unit	test-data-processing.ts	should fetch crime data for specific GeoIDs	Tests fetchCrimeData with valid GeoID array and validates crime data retrieval	result defined, result is array, mockFetchCrimeData called with mockSupabase and ['36061019500']
Unit	test-data-processing.ts	should return null for empty GeoIDs array	Tests fetchCrimeData with empty GeoIDs array	result = null
Unit	test-data-processing.ts	should return null for null GeoIDs	Tests fetchCrimeData with null GeoIDs parameter	result = null

Unit	test-data-processing.ts	should handle database errors gracefully	Tests fetchCrimeData database error handling	result = null
Unit	test-data-processing.ts	should handle exceptions gracefully	Tests fetchCrimeData exception handling	result = null
Unit	test-data-processing.ts	should fetch foot traffic data for specific GeoIDs	Tests fetchFootTrafficData with valid GeoID array and validates traffic data retrieval	result defined, result is array, mockFetchFootTrafficData called with mockSupabase and ['36061019500']
Unit	test-data-processing.ts	should return null for empty GeoIDs array	Tests fetchFootTrafficData with empty GeoIDs array	result = null
Unit	test-data-processing.ts	should return null for undefined GeoIDs	Tests fetchFootTrafficData with undefined GeoIDs parameter	result = null
Unit	test-data-processing.ts	should handle database errors gracefully	Tests fetchFootTrafficData database error handling	result = null
Unit	test-data-processing.ts	should handle exceptions gracefully	Tests fetchFootTrafficData exception handling	result = null
Unit	test-data-processing.ts	should validate data with all required fields	Tests validateDatabaseData with complete data containing all required fields (zones, ethnicity, demographics, income)	result = true, mockValidateDatabaseData called
Unit	test-data-processing.ts	should fail validation when zones data is missing	Tests validateDatabaseData with empty zones array	result = false
Unit	test-data-processing.ts	should fail validation when zones is null	Tests validateDatabaseData with null zones	result = false
Unit	test-data-processing.ts	should pass validation with missing optional data	Tests validateDatabaseData with zones present but optional data (ethnicity, demographics, income) null	result = true
Unit	test-data-processing.ts	should handle partial missing optional data	Tests validateDatabaseData with zones and some ethnicity data but	result = true

			demographics null and income empty	
Unit	test-data-processing.ts	should handle fetchAllData with network exception	Tests fetchAllData error handling with network failure exception	Throws error with 'Database fetch failed' message
Unit	test-data-processing.ts	should handle multiple GeoIDs for crime data	Tests fetchCrimeData with multiple GeoIDs and validates bulk crime data retrieval	mockFetchCrimeData called with mockSupabase and ['36061019500', '36061019600', '36061019700'], result equals mockData
Unit	test-data-processing.ts	should handle multiple GeoIDs for foot traffic data	Tests fetchFootTrafficData with multiple GeoIDs and validates bulk traffic data retrieval	mockFetchFootTrafficData called with mockSupabase and ['36061019500', '36061019600'], result equals mockData
Unit	test-demographic-scoring.ts	should calculate ethnicity percentages correctly	Tests calculateDemographicPercentages with mock ethnicity data (Korean 20%, Chinese 15%) and validates ethnicity percentage calculation	result.ethnicPercent defined, result.ethnicPercent['36061019500'] > 0 and ≤ 1
Unit	test-demographic-scoring.ts	should calculate gender percentages correctly	Tests calculateDemographicPercentages with mock demographics data (45% male) and validates gender percentage calculation	result.genderPercent defined, result.genderPercent['36061019500'] = 0.45
Unit	test-demographic-scoring.ts	should calculate age percentages correctly	Tests calculateDemographicPercentages with mock age data (15% + 10% in age range 25-35) and validates age percentage calculation	result.agePercent defined, result.agePercent['36061019500'] = 0.25
Unit	test-demographic-scoring.ts	should calculate income percentages correctly	Tests calculateDemographicPercentages with mock income data and validates income percentage calculation	result.incomePercent defined, result.incomePercent['36061019500'] > 0
Unit	test-demographic-scoring.ts	should handle empty data arrays	Tests calculateDemographicPercentages with empty input arrays	result.ethnicPercent = {}, result.genderPercent = {}, result.agePercent = {}, result.incomePercent = {}

			and validates empty result handling	
Unit	test-demographic-scoring.ts	should prevent ethnicity overcounting	Tests calculateDemographicPercentages with overlapping ethnicity data (Asian categories) and validates deduplication logic	result.ethnicPercent['36061019500'] ≤ 1.0 (prevents overcounting)
Unit	test-demographic-scoring.ts	should find maximum percentages across all tracts	Tests findMaxPercentages with percentage data across multiple tracts and validates maximum calculation	result.maxEthnicPct = 1, result.maxGenderPct = 1, result.maxAgePct = 1, result.maxIncomePct = 1 (default minimum)
Unit	test-demographic-scoring.ts	should handle empty percentage objects	Tests findMaxPercentages with empty percentage objects and validates default handling	result.maxEthnicPct = 1, result.maxGenderPct = 1, result.maxAgePct = 1, result.maxIncomePct = 1
Unit	test-demographic-scoring.ts	should calculate enhanced demographic score	Tests calculateEnhancedDemographicScore with mock percentages and validates score calculation	result > 0 and ≤ 1
Unit	test-demographic-scoring.ts	should return zero for tract with no demographic data	Tests calculateEnhancedDemographicScore with empty percentages for non-existent tract and validates zero score	result = 0
Unit	test-demographic-scoring.ts	should handle advanced demographic scoring with weights	Tests calculateEnhancedDemographicScore with custom demographic weights (ethnicity: 0.4, gender: 0.3, age: 0.2, income: 0.1)	result > 0 and ≤ 1
Unit	test-demographic-scoring.ts	should evaluate greater than conditions correctly	Tests evaluateCondition with greater than operators on mock tract data	'population > 500' = true, 'population > 1500' = false
Unit	test-demographic-scoring.ts	should evaluate less than conditions correctly	Tests evaluateCondition with less than operators on mock tract data	'crime_rate < 3.0' = true, 'crime_rate < 2.0' = false

Unit	test-demographic-scoring.ts	should evaluate equals conditions correctly	Tests evaluateCondition with equality operators on mock tract data	'age_median == 35' = true, 'age_median == 30' = false
Unit	test-demographic-scoring.ts	should evaluate greater than or equal conditions correctly	Tests evaluateCondition with greater than or equal operators on mock tract data	'income >= 75000' = true, 'income >= 80000' = false
Unit	test-demographic-scoring.ts	should handle invalid condition formats	Tests evaluateCondition with invalid condition strings and validates error handling	'invalid condition' = false, 'population' = false
Unit	test-demographic-scoring.ts	should handle missing tract data	Tests evaluateCondition with missing fields in tract data and validates error handling	'missing_field > 100' = false, 'population > invalid' = false
Unit	test-index-module.ts	should handle OPTIONS preflight requests	Tests handleRequest with OPTIONS method for CORS preflight and validates proper CORS response	response.status = 200, Access-Control-Allow-Origin = '*', Access-Control-Allow-Methods contains 'POST', body = 'ok'
Unit	test-index-module.ts	should accept POST requests	Tests handleRequest with POST method and valid JSON body	response.status = 200
Unit	test-index-module.ts	should reject non-POST requests	Tests handleRequest with GET method and validates method rejection	response.status = 405, body.error = 'Method not allowed'
Unit	test-index-module.ts	should include required CORS headers	Tests handleRequest OPTIONS response for required CORS headers	Access-Control-Allow-Origin = '*', headers contain 'authorization' and 'content-type', Max-Age = '86400'
Unit	test-index-module.ts	should include CORS headers on error responses	Tests handleRequest error response includes CORS headers	Access-Control-Allow-Origin = '*', Content-Type = 'application/json'
Unit	test-index-module.ts	should handle valid JSON requests	Tests handleRequest with valid JSON body containing weights, ethnicities, and topN	response.status = 200, body has 'zones', 'total_zones_found', message = 'success'
Unit	test-index-module.ts	should handle empty JSON requests	Tests handleRequest with empty JSON object	response.status = 200
Unit	test-index-module.ts	should handle malformed JSON gracefully	Tests handleRequest with malformed JSON body ('invalid json {')	response.status = 500, body.error = 'Internal server error', body.timestamp defined

Unit	test-index-module.ts	should handle missing request body	Tests handleRequest with POST method but no request body	response.status = 500
Unit	test-index-module.ts	should create comprehensive debug info	Tests createDebugInfo with mock input, zones (100), and processed zones (25)	debug.received_ethnicities = ['korean', 'chinese'], debug.received_genders = ['female'], debug.total_zones_before_filters = 100, debug.total_zones_after_filters = 25, debug.zones_filtered_out = 75
Unit	test-index-module.ts	should handle empty input gracefully	Tests createDebugInfo with empty arrays and validates empty input handling	debug.received_ethnicities = [], debug.total_zones_before_filters = 0, debug.zones_filtered_out = 0
Unit	test-index-module.ts	should validate required environment variables	Tests validateEnvironment with empty process.env and validates missing variable detection	result.valid = false, result.missing contains 'SUPABASE_URL' and 'SUPABASE_ANON_KEY'
Unit	test-index-module.ts	should pass validation with all required variables	Tests validateEnvironment with SUPABASE_URL and SUPABASE_ANON_KEY set	result.valid = true, result.missing.length = 0
Unit	test-index-module.ts	should process complete request successfully	Tests processCompleteRequest with valid request containing weights, ethnicities, and topN	result has 'zones', 'total_zones_found', 'debug' properties, result.zones.length = 2
Unit	test-index-module.ts	should handle processing errors gracefully	Tests processCompleteRequest with null request and validates error handling	result has 'error' and 'timestamp' properties
Unit	test-index-module.ts	should maintain consistent error response format	Tests handleRequest error response format consistency with malformed body	body has 'error' and 'timestamp' properties, both are strings
Unit	test-index-module.ts	should include timestamp in error responses	Tests handleRequest error response includes valid timestamp	body.timestamp defined, new Date(body.timestamp) is valid Date instance
Unit	test-index-module.ts	should return consistent	Tests handleRequest success response	body has 'zones', 'total_zones_found', 'message', zones is array,

		success response structure	structure with valid POST request	total_zones_found is number
Unit	test-index-module.ts	should set correct content type headers	Tests handleRequest response Content-Type header	Content-Type = 'application/json'
Unit	test-scoring-helpers.ts	should process zones	Tests processZones with zone data containing GEOID '36061019500' and validates zone processing functionality	result.length = 1, result[0].geoid = '36061019500', mockProcessZones called
Unit	test-scoring-helpers.ts	should add foot traffic data	Tests addFootTrafficDataToTopZones with mock zones and validates foot traffic data enrichment	mockAddFootTrafficDataToTopZones called with database object and zones array
Unit	test-scoring-helpers.ts	should add crime data	Tests addCrimeDataToTopZones with mock zones and validates crime data enrichment	mockAddCrimeDataToTopZones called with database object and zones array
Unit	test-utils-module.ts	VALID_GENDERS should contain male and female	Tests VALID_GENDERS constant contains correct gender values	VALID_GENDERS = ['male', 'female']
Unit	test-utils-module.ts	VALID_TIME_PERIODS and DEFAULT_TIME_PERIODS should match expected values	Tests time period constants contain correct values for morning, afternoon, evening	VALID_TIME_PERIODS = ['morning', 'afternoon', 'evening'], DEFAULT_TIME_PERIODS = ['morning', 'afternoon', 'evening']
Unit	test-utils-module.ts	DEFAULT_CRIME_YEARS should contain correct years	Tests DEFAULT_CRIME_YEARS constant contains historical and predicted crime years	DEFAULT_CRIME_YEARS = ['year_2020', 'year_2021', 'year_2022', 'year_2023', 'year_2024', 'pred_2025', 'pred_2026', 'pred_2027']
Unit	test-utils-module.ts	returns correct JSON response with status	Tests jsonResponse function with custom data and status code (201)	response.status = 201, Content-Type = 'application/json', parsed JSON equals input data
Unit	test-utils-module.ts	returns 200 by default	Tests jsonResponse function default status code when no status provided	response.status = 200
Unit	test-utils-module.ts	returns first N entries from an object	Tests getSampleEntries function with object {a:1, b:2, c:3, d:4} requesting first 2 entries	Object.keys(result) = ['a', 'b']

Unit	test-utils-module.ts	handles empty or invalid input	Tests getSampleEntries function with null, array, and string inputs	getSampleEntries(null) = {}, getSampleEntries([]) = {}, getSampleEntries('test') = {}
Unit	test-utils-module.ts	returns correct boroughs based on geoId	Tests getBoroughName function with valid NYC borough GeoIDs	GeoID '3606100000000' = 'Manhattan', '3600500000000' = 'Bronx', '3604700000000' = 'Brooklyn', '3608100000000' = 'Queens', '3608500000000' = 'Staten Island'
Unit	test-utils-module.ts	returns Unknown for invalid or unknown geoIds	Tests getBoroughName function with invalid GeoID inputs	getBoroughName('') = 'Unknown', getBoroughName('999999') = 'Unknown', getBoroughName(null) = 'Unknown'
Unit	test-utils-module.ts	normalizes within 0-100 range	Tests normalizeScore function with values below, within, and above 0-100 range	normalizeScore(-10) = 0, normalizeScore(50) = 50, normalizeScore(120) = 100
Unit	test-utils-module.ts	handles null, undefined, NaN gracefully	Tests normalizeScore function with invalid input types	normalizeScore(null) = 0, normalizeScore(undefined) = 0, normalizeScore('bad') = 0
Unit	test-utils-module.ts	returns value within min/max bounds	Tests clamp function with values within, below, and above specified bounds (0-10)	clamp(5, 0, 10) = 5, clamp(-1, 0, 10) = 0, clamp(100, 0, 10) = 10
Unit	test-utils-module.ts	correctly rounds to given decimal places	Tests roundTo function with various numbers and decimal place specifications	roundTo(3.14159, 2) = 3.14, roundTo(3.14159, 0) = 3, roundTo(3.9999, 2) = 4
Unit	test-utils-module.ts	handles invalid inputs	Tests roundTo function with null, undefined, and string inputs	roundTo(null, 2) = 0, roundTo(undefined, 2) = 0, roundTo('bad', 2) = 0
Unit	test-validation-module.ts	should validate complete valid request	Tests validateRequestBody with complete valid request containing weights, ethnicities, genders, age range, income range, and topN	result.weights.length = 3, result.ethnicities = ['korean', 'chinese'], result.genders = ['male', 'female'], result.ageRange = [25, 65], result.topN = 15
Unit	test-validation-module.ts	should apply defaults for missing fields	Tests validateRequestBody with empty object and	result.weights = [], result.ethnicities = [], result.genders = [], result.ageRange = [0, 100],

			validates default value assignment	result.incomeRange = [0, 250000], result.rentRange = [0, Infinity], result.topN = 10, result.timePeriods = ['morning', 'afternoon', 'evening']
Unit	test-validation-module.ts	should filter invalid weights	Tests validateRequestBody with mix of valid and invalid weights (missing id, invalid value, negative value)	result.weights.length = 2, result.weights[0].id = 'foot_traffic', result.weights[1].id = 'demographic'
Unit	test-validation-module.ts	should clamp age ranges to valid bounds	Tests validateRequestBody with age range [-10, 150] and validates clamping to valid bounds	result.ageRange = [0, 100]
Unit	test-validation-module.ts	should filter invalid genders and time periods	Tests validateRequestBody with valid and invalid genders/time periods (includes 'other' gender and 'night' period)	result.genders = ['male', 'female'], result.timePeriods = ['morning', 'evening']
Unit	test-validation-module.ts	should validate correct demographic scoring structure	Tests validateDemographicScoring with valid weights object containing ethnicity, gender, age, and income weights	validateDemographicScoring returns true
Unit	test-validation-module.ts	should reject missing weights object	Tests validateDemographicScoring with empty object (no weights property)	validateDemographicScoring returns false
Unit	test-validation-module.ts	should reject invalid weight values	Tests validateDemographicScoring with invalid weight values (>1, <0, string)	validateDemographicScoring returns false
Unit	test-validation-module.ts	should reject missing required weight keys	Tests validateDemographicScoring with incomplete weights object (missing age and income)	validateDemographicScoring returns false
Unit	test-validation-module.ts	should handle null/undefined input	Tests validateDemographicScoring with null and undefined inputs	validateDemographicScoring(null) = false, validateDemographicScoring(undefined) = false

Unit	test-validation-module.ts	should validate correct weight structure	Tests validateWeightStructure with valid weight array and validates structure validation	result.valid = true, result.errors.length = 0, result.totalWeight = 90
Unit	test-validation-module.ts	should detect duplicate weight IDs	Tests validateWeightStructure with duplicate 'foot_traffic' weight IDs	result.valid = false, result.errors contains 'Duplicate weight id: foot_traffic'
Unit	test-validation-module.ts	should reject invalid weight IDs and values	Tests validateWeightStructure with invalid weight IDs and out-of-range values	result.valid = false
Unit	test-validation-module.ts	should handle non-array input	Tests validateWeightStructure with string input instead of array	result.valid = false, result.errors contains 'Weights must be an array'
Unit	test-validation-module.ts	should sanitize valid string arrays	Tests sanitizeStringArray with valid string array ['korean', 'chinese']	sanitizeStringArray returns ['korean', 'chinese']
Unit	test-validation-module.ts	should filter non-string values and trim whitespace	Tests sanitizeStringArray with mixed array containing strings with whitespace, numbers, and null values	sanitizeStringArray returns ['korean', 'chinese'] (filtered and trimmed)
Unit	test-validation-module.ts	should remove empty strings and enforce length limits	Tests sanitizeStringArray with empty strings and strings exceeding length limit (maxLength=10)	sanitizeStringArray returns ['short'] (long string filtered out)
Unit	test-validation-module.ts	should handle non-array input	Tests sanitizeStringArray with null and string inputs	sanitizeStringArray(null) = [], sanitizeStringArray('string') = []
Unit	test-validation-module.ts	should validate correct numeric ranges	Tests validateNumericRange with valid range [25, 65] within bounds [0, 100]	validateNumericRange returns [25, 65]
Unit	test-validation-module.ts	should reject invalid range formats	Tests validateNumericRange with single value array and string input	validateNumericRange([25], 0, 100) = null, validateNumericRange('string', 0, 100) = null
Unit	test-validation-module.ts	should reject invalid values or order	Tests validateNumericRange with reversed range,	validateNumericRange([65, 25], 0, 100) = null, validateNumericRange([NaN, 30], 0, 100) = null,

			NaN values, and out-of-bounds values	validateNumericRange([0, 150], 0, 100) = null
Unit	test-validation-module.ts	should allow edge bounds	Tests validateNumericRange with edge case values at minimum and maximum bounds	validateNumericRange([0, 0], 0, 100) = [0, 0], validateNumericRange([100, 100], 0, 100) = [100, 100]
Integration	test-filter.ts	should validate and apply filter parameters	Tests complete filter validation and application with weights, ethnicities, genders, age range, income range, rent range, and topN parameters	validatedData.ethnicities contains 'korean', validatedData.genders = ['male', 'female'], validatedData.ageRange = [25, 65], processedZones.length = 1, processedZones[0].geoid = '36061019500', processedZones[0].demographic_match_pct = 25.3
Integration	test-filter.ts	should handle empty filters gracefully	Tests filter handling with empty ethnicities and genders arrays, applying default values	result.ethnicities.length = 0, result.genders.length = 0, data.zones.length = 2
Integration	test-filter.ts	should apply ethnicity filter correctly	Tests ethnicity filtering with Korean ethnicity data and demographic percentage calculations	percentages.ethnicPercent['36061019500'] = 0.253, zones[0].demographic_match_pct = 25.3, zones[1].demographic_match_pct = 18.7
Integration	test-filter.ts	should apply rent range filter correctly	Tests rent range filtering (3000-4000) with zone data containing different rent values	result.length = 2, result[0].avg_rent = 3200, result[1].avg_rent = 3800, zone with rent $4500 filtered out
Integration	test-filter.ts	should apply demographic scoring correctly	Tests enhanced demographic scoring with Korean ethnicity, male gender, age range 25-40, and custom demographic weights	calculateEnhancedDemographicScore returns 0.85, called with '36061019500', demographicInput, mockPercentages, {}
Integration	test-filter.ts	should apply multiple filters together	Tests combined filtering with Korean ethnicity, rent range 3000-5000, age range 25-45, and multiple weight factors	validated.ethnicities contains 'korean', validated.rentRange = [3000, 5000], validated.ageRange = [25, 45], zones[0].demographic_match_pct = 22.1, zones[0].avg_rent = 3500
Integration	test-filter.ts	should handle restrictive filter combinations	Tests restrictive filtering with high rent (5000-6000), young	result.length = 0

			age (20-25), and high income (200000-300000) that results in no matches	
Integration	test-filter.ts	should handle invalid filter ranges	Tests error handling for invalid filter ranges where maximum is less than minimum	validateRequestBody throws 'Invalid rent range' error
Integration	test-filter.ts	should handle invalid ethnicity values	Tests error handling for unsupported ethnicity codes	validateRequestBody throws 'Unsupported ethnicity codes: invalid_ethnicity, another_invalid'
Integration	test-filter.ts	should handle extreme filter values	Tests handling of extreme filter values with automatic clamping to valid ranges	result.rentRange[0] ≥ 0, result.ageRange[0] ≥ 0, result.ageRange[1] ≤ 100, result.incomeRange[1] ≤ 1000000
Integration	test-filter.ts	should process complex filters efficiently	Tests performance of complex filtering with multiple weights, ethnicities, demographics, and large dataset (100 zones)	isValid = true, zones.length = 25, executionTime < 100ms
Integration	test-filter.ts	should return properly formatted response	Tests edge function response format validation with all required fields and proper data types	mockResponse.zones defined and is array, total_zones_found is number, top_zones_returned is number, filters_applied is object, debug is object, zone.geoid is string, zone.resilience_score is number, zone.custom_score is number
Integration	test-zone-search.ts	should make basic request and return zones	Tests basic zone search functionality with realistic database data and zone processing, validates response structure and data types	isValid = true, zones is array with length > 0, zone.geoid matches /^\d{11}$/, zone.resilience_score ≥ 0 and ≤ 100, zone.custom_score ≥ 0 and ≤ 100, zone has properties: geoid, resilience_score, custom_score, avg_rent, tract_name, display_name
Integration	test-zone-search.ts	should validate response structure with multiple zones	Tests response structure consistency across multiple zones and validates proper sorting by custom_score	All zones have consistent structure with valid GEOIDs and score ranges, zones properly sorted by custom_score (highest first)

Integration	test-zone-search.ts	should handle rent range filtering	Tests rent range filtering (3000-4000) with zone data, validates filtering logic and range validation	result.length = 2, validated.rentRange = [3000, 4000], all zones have avg_rent ≥ 3000 and ≤ 4000
Integration	test-zone-search.ts	should handle demographic filtering integration	Tests demographic filtering integration with Korean ethnicity and age range 25-45, validates demographic percentage calculations	result[0].demographic_match_pct = 15.2, result[0].age_match_pct = 75.0, result[0].demographic_score = 85
Integration	test-zone-search.ts	should handle empty results gracefully	Tests handling of search queries with unrealistic criteria that return no results	result.length = 0
Integration	test-zone-search.ts	should handle database errors properly	Tests database connection error handling during data fetching	fetchAllData throws 'Database connection failed' error
Integration	test-zone-search.ts	should handle invalid filter parameters	Tests validation error handling for invalid parameter types (rentRange as string instead of array)	validateRequestBody throws 'rentRange must be an array of two numbers' error
Integration	test-zone-search.ts	should handle validation errors gracefully	Tests database data validation with empty zones array	isValid = false
Integration	test-zone-search.ts	should complete searches within reasonable time	Tests search performance with single zone data and measures execution time	zones.length = 1, executionTime < 100ms
Integration	test-zone-search.ts	should handle large result sets efficiently	Tests performance with large dataset (100 zones) and measures processing efficiency	result.length = 100, executionTime < 200ms
Integration	test-zone-search.ts	should return properly formatted edge function response	Tests complete edge function flow with data fetching, zone processing, and timeline data enrichment	processedZones is array with length = 1, zone properties are correct types (geoid: string, resilience_score: number, custom_score: number, tract_name: string, display_name: string), zone has crime_timeline and foot_traffic_timeline properties
Integration	test-zone-search.ts	should handle demographic	Tests demographic scoring integration with Korean ethnicity,	percentages.ethnicPercent['36061019500'] = 0.15, percentages.genderPercent[

		scoring response format	male gender, and custom demographic weights	'36061019500'] = 0.52, calculateEnhancedDemographicScore returns 0.78
Integration	test-zone-search.ts	should properly enrich zones with timeline data	Tests zone enrichment integration with crime and foot traffic timeline data, validates trend direction assignment	mockZones[0] has crime_timeline, foot_traffic_timeline, crime_trend_direction, and foot_traffic_trend_direction properties, crime_trend_direction = 'decreasing', foot_traffic_trend_direction = 'increasing'

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
Database	Database operations, constraints, validation & CRUD operations	55	55	0	100%
Unit	Individual function testing, data processing, validation, utilities	97	97	0	100%
Integration	End-to-end filtering, zone search, complete workflows	25	25	0	100%
Total		**177**	**177**	**0**	**100%**