SQE Final Project

Project Title: Comprehensive Quality Engineering for the Open-Source Crater Application

Testers: Uzair Majeed 23i-3063, Hussnain Haider 23i-0695, Faez Ahmed 23i-0598

Section: SE-B

Integration Testing Report (IEEE 829-2008 Standard)

Index of Test Files

42.	ExpenseTest.txt
43.	InvoiceItemTest.txt
44.	InvoiceTest.txt
45.	ItemTest.txt
46.	PaymentMethodTest.txt
47.	PaymentTest.txt
48.	RecurringInvoiceTest.txt
49.	SettingTest.txt
50.	TaxTest.txt
51.	TaxTypeTest.txt
52.	UnitTest.txt
53.	UserTest.txt

File: AddressTest.txt

Test ID	TC-AT-001
Title	Verify Address Entity Association with User
Objective	Validate that an Address created in the system is properly linked to a valid User entity at the database level.
Preconditions	- Application environment is running. - Database has been seeded using DatabaseSeeder and DemoSeeder. - Address, User entities and their relationships are configured in the ORM.
Steps	Step 1: Seed database using DatabaseSeeder and DemoSeeder. Step 2: Create an Address entity using the factory, associating it to a User. Step 3: Check if the Address's user association exists in the database.
Test Data	- Address entity created using Address::factory()->forUser()->create(). - Expected: The Address entity references an existing User.
Expected Result	The created Address entity has a valid existing User association, confirming the belongs-to relationship is functional and correctly mapped.
Actual Result	The created Address entity has a valid existing User association, confirming the belongs-to relationship is functional and correctly mapped.
Status	Pass
Severity	High

Test ID	TC-AT-002
Title	Verify Address Entity Association with Country
Objective	Ensure that an Address created in the system correctly references a valid Country entity, validating foreign key and ORM relationship integrity.
Preconditions	- Application environment is running. - Database has been seeded using DatabaseSeeder and DemoSeeder. - Address and Country entities exist and are related in the database schema.
Steps	Step 1: Seed database using DatabaseSeeder and DemoSeeder. Step 2: Create an Address entity using the factory. Step 3: Check if the Address's country association exists.
Test Data	- Address entity created using Address::factory()->create(). - Expected: The Address entity references an existing Country entity.
Expected Result	The Address entity has a valid association with a Country entity, confirming the belongs-to relationship is functional and data integrity is maintained.
Actual Result	The Address entity has a valid association with a Country entity, confirming the belongs-to relationship is functional and data integrity is maintained.
Status	Pass
Severity	Medium

Test ID	TC-AT-003
Title	Verify Address Entity Association with Customer
Objective	Confirm that an Address associated with a Customer is properly linked and the relationship is retrievable at an integrated system level.
Preconditions	- Application environment is running. - Database is seeded using DatabaseSeeder and DemoSeeder. - Address and Customer entities with correct relationship mapping are present.
Steps	Step 1: Seed database using DatabaseSeeder and DemoSeeder. Step 2: Create an Address entity using the factory and associate it to a Customer. Step 3: Verify that the customer relationship for the created Address exists.

Test Data	- Address entity created using Address::factory()->forCustomer()->create(). - Expected: The Address entity references an existing Customer.
Expected Result	The created Address entity has a valid association with a Customer, confirming relationship is correctly integrated and retrievable.
Actual Result	The created Address entity has a valid association with a Customer, confirming relationship is correctly integrated and retrievable.
Status	Pass
Severity	Medium

File: Admin\BackupTest.txt

Test ID	TC-Adm-BT-001
Title	Retrieve Existing Backups via API for a Specific Storage Disk
Objective	Validate the integration between the backup retrieval API endpoint and the storage disk, ensuring backups can be listed for a selected disk with correct authentication and company context.
Preconditions	- Application running with all backend services active - Database seeded with DatabaseSeeder and DemoSeeder - At least one user (ID:1) and default company present - At least one FileDisk record exists and is set as default - API authentication established using Sanctum, acting as user ID:1
Steps	Step 1: Create a default FileDisk record for testing Step 2: Send a GET request to /api/v1/backups with disk and file_disk_id parameters Step 3: Verify the response status is HTTP 200 OK
Test Data	Inputs: - GET request to /api/v1/backups with parameters: - disk = [the driver of the created FileDisk] - file_disk_id = [ID of the created FileDisk] - Company header set to user’s first company ID - Sanctum authentication for user ID:1 Expected values: - HTTP 200 OK status upon successful retrieval
Expected Result	Backups list is successfully retrieved for the specified storage disk, response returns HTTP 200 OK, confirming correct interaction between API and backup storage system.
Actual Result	Backups list is successfully retrieved for the specified storage disk, response returns HTTP 200 OK, confirming correct interaction between API and backup storage system.
Status	Pass
Severity	High

Test ID	TC-Adm-BT-002
Title	Initiate New Full Backup Creation and Validate Background Job Dispatch
Objective	Verify that creating a new backup via API successfully triggers the background job dispatcher, stores the request, and returns updated backup list from storage disk.
Preconditions	- Application running with all backend services active - Database seeded with DatabaseSeeder and DemoSeeder - At least one user (ID:1), default company, and storage disk present - API authentication established using Sanctum, acting as user ID:1 - Queue system enabled and faked for testing
Steps	Step 1: Set up a new FileDisk record for backup target Step 2: Send a POST request to /api/v1/backups with full backup option payload Step 3: Assert that CreateBackupJob is dispatched to the queue Step 4: Send a GET request to /api/v1/backups to retrieve the updated backup list for the disk Step 5: Confirm response is HTTP 200 and contains the "local" disk in the “disks” list

Test Data	Inputs: - POST request to /api/v1/backups with payload: - option = 'full' - file_disk_id = [ID of created FileDisk] - Subsequent GET request to /api/v1/backups with disk and file_disk_id parameters - Company header set to user’s first company ID - Sanctum authentication for user ID:1 Expected values: - CreateBackupJob is dispatched to the queue - GET response returns backups list, including expected storage disks (e.g., "local") - HTTP status 200
Expected Result	A full backup job is successfully queued, and subsequently, the backup list endpoint includes the "local" disk, reflecting new backup availability; all API responses return HTTP 200 OK.
Actual Result	A full backup job is successfully queued, and subsequently, the backup list endpoint includes the "local" disk, reflecting new backup availability; all API responses return HTTP 200 OK.
Status	Pass
Severity	High

File: Admin\CompanySettingTest.txt

Test ID	TC-Adm-CST-001
Title	Retrieve Logged-In User Profile Information
Objective	Validate the integration between authentication middleware and the user profile API endpoint, confirming that the authenticated user's profile details are accessible.
Preconditions	- Application server is running. - Database is seeded with required user/company data. - API is accessible. - User is authenticated via Sanctum; company context provided in request headers.
Steps	Step 1: Send GET request to 'api/v1/me' endpoint with authenticated session. Step 2: Validate that the HTTP response status is OK (200).
Test Data	- No input payload (GET request). - Authenticated user context.
Expected Result	A successful (HTTP 200) response is returned containing the authenticated user's profile information.
Actual Result	A successful (HTTP 200) response is returned containing the authenticated user's profile information.
Status	Pass
Severity	High

Test ID	TC-Adm-CST-002
Title	Ensure Profile Update Endpoint Uses Proper Form Request Validation
Objective	Verify that the controller action responsible for updating user profiles integrates correctly with the ProfileRequest form request for input validation.
Preconditions	- Application server is running. - Database seeded with users. - API accessible.
Steps	Step 1: Assert that the CompanyController@updateProfile method uses the ProfileRequest for input validation.
Test Data	- Controller: CompanyController. - Method: updateProfile. - Form Request: ProfileRequest.
Expected Result	The updateProfile controller action is bound to ProfileRequest, enforcing robust profile validation during updates.
Actual Result	The updateProfile controller action is bound to ProfileRequest, enforcing robust profile validation during updates.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CST-003
Title	Update User Profile Information and Persist Changes to Database
Objective	Verify integration between the profile update API, controller logic, validation layer, and database to confirm that profile updates are correctly processed and saved for the authenticated user.
Preconditions	- Application server is running. - Database seeded with initial user data. - API accessible. - User is authenticated via Sanctum.

Steps	Step 1: Send PUT request to 'api/v1/me' with new profile data. Step 2: Confirm the HTTP response status is OK (200). Step 3: Assert user's name and email are updated in the database.
Test Data	- Request Payload: - name: 'John Doe' - password: 'admin@123' - email: 'admin@crater.in' - Expected values: - Database contains 'users' record with name 'John Doe' and email 'admin@crater.in'
Expected Result	API responds with HTTP 200; user's profile fields (name, email) are updated and persisted to the database.
Actual Result	API responds with HTTP 200; user's profile fields (name, email) are updated and persisted to the database.
Status	Pass
Severity	High

Test ID	TC-Adm-CST-004
Title	Ensure Company Update Endpoint Applies Proper Validation
Objective	Validate that the CompanyController@updateCompany method integrates with CompanyRequest to enforce correct business rules during company updates.
Preconditions	- Application server is running. - Database seeded with companies. - API accessible.
Steps	Step 1: Assert the CompanyController@updateCompany uses CompanyRequest for validation.
Test Data	- Controller: CompanyController. - Method: updateCompany. - Form Request: CompanyRequest.
Expected Result	Controller action for company update is bound to CompanyRequest, ensuring all company updates are validated according to business rules.
Actual Result	Controller action for company update is bound to CompanyRequest, ensuring all company updates are validated according to business rules.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CST-005
Title	Update Company Profile and Address Information via API
Objective	Validate the workflow for updating company details through the API, including proper validation, persistence in both 'companies' and 'addresses' tables, and correct integration among request layer, controller, and database.
Preconditions	- Application server is running. - Database seeded with company and address data. - API accessible. - User authenticated with company context.
Steps	Step 1: Send PUT request to 'api/v1/company' with updated company and address details. Step 2: Assert HTTP response is OK. Step 3: Verify 'companies' table has the updated name. Step 4: Verify 'addresses' table has updated country_id.

Test Data	- Request Payload: - name: 'XYZ' - country_id: 2 - state: 'city' - city: 'state' - address_street_1: 'test1' - address_street_2: 'test2' - phone: '1234567890' - zip: '112233' - address: ['country_id' => 2] - Expected database records: - 'companies' table: name 'XYZ' - 'addresses' table: country_id 2
Expected Result	API responds with HTTP 200; company and address information are updated and persisted to their respective tables.
Actual Result	API responds with HTTP 200; company and address information are updated and persisted to their respective tables.
Status	Pass
Severity	High

Test ID	TC-Adm-CST-006
Title	Update Company Settings with Various Configuration Options
Objective	Validate the integration of the settings API, controller, and database—ensuring that multiple company settings fields are updated together and persisted correctly.
Preconditions	- Application server is running. - Database seeded with company settings. - API accessible. - User authenticated.
Steps	Step 1: Send POST request to '/api/v1/company/settings' with a full settings payload. Step 2: Validate HTTP response is OK and JSON contains 'success' = true. Step 3: Assert each key-value pair exists in the 'company_settings' table.
Test Data	- Settings provided in request: - currency: 1 - time_zone: 'Asia/Kolkata' - language: 'en' - fiscal_year: '1-12' - carbon_date_format: 'Y/m/d' - moment_date_format: 'YYYY/MM/DD' - notification_email: 'noreply@crater.in' - notify_invoice_viewed: 'YES' - notify_estimate_viewed: 'YES' - tax_per_item: 'YES' - discount_per_item: 'YES' - Expected database records: Each setting option is stored with the corresponding value in 'company_settings' table.
Expected Result	API returns HTTP 200 with JSON success; all provided settings are updated and persisted in 'company_settings' table.
Actual Result	API returns HTTP 200 with JSON success; all provided settings are updated and persisted in 'company_settings' table.
Status	Pass
Severity	High

Test ID	TC-Adm-CST-007
Title	Update Notification Email Setting Individually

Objective	Verify that individual company settings (notification email) can be updated independently via the API and that changes persist correctly.
Preconditions	- Application server is running. - Database seeded with company settings. - API accessible. - Authenticated session with company context.
Steps	Step 1: Send POST request to '/api/v1/company/settings' with only the notification_email setting. Step 2: Confirm HTTP response is OK with JSON 'success' = true. Step 3: Assert the 'company_settings' entry for notification_email has correct value.
Test Data	- Request Payload: - notification_email: 'noreply@crater.in' - Expected database record: - 'company_settings' table: option 'notification_email', value 'noreply@crater.in'
Expected Result	API responds with HTTP 200 and JSON success; notification_email setting is updated in the database.
Actual Result	API responds with HTTP 200 and JSON success; notification_email setting is updated in the database.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CST-008
Title	Prevent Changing Currency Setting When Transactions Exist
Objective	Ensure business logic prevents changes to the company currency after financial transactions are present; validate error message and database integrity on attempted change.
Preconditions	- Application server is running. - Database seeded with company settings and transactional data (invoice, taxes, items). - API accessible. - Authenticated user/company context.
Steps	Step 1: Send POST request to '/api/v1/company/settings' to set currency to 1. Step 2: Confirm successful response and currency value is set. Step 3: Simulate creation of invoice and linked items/taxes. Step 4: Attempt to update currency to a different value (2) via settings API. Step 5: Confirm HTTP response is OK, JSON contains 'success' = false and correct error message. Step 6: Assert that the currency value in the database has not changed.
Test Data	- Initial settings update request: currency: 1 - Creation of invoice with associated taxes/items - Attempted settings update request: currency: 2 - Expected error: 'Cannot update company currency after transactions are created.' - Database value: currency remains as 1
Expected Result	API returns error indicating currency change is prohibited when transactions exist; database retains original currency value.
Actual Result	API returns error indicating currency change is prohibited when transactions exist; database retains original currency value.
Status	Pass
Severity	High

Test ID	TC-Adm-CST-009
Title	Retrieve Current Company Notification and General Settings

Objective	Verify that the settings API endpoint correctly returns current configuration values for the company, integrating with the database and authentication system.
Preconditions	- Application server is running. - Database seeded with company settings. - API accessible. - Authenticated company context.
Steps	Step 1: Send GET request to '/api/v1/company/settings' with relevant settings query parameters. Step 2: Assert HTTP response status is OK and settings data is returned.
Test Data	- GET parameters: settings[] = - currency - time_zone - language - fiscal_year - carbon_date_format - moment_date_format - notification_email - notify_invoice_viewed - notify_estimate_viewed - tax_per_item - discount_per_item - Expected: HTTP 200 OK response with settings data for requested keys.
Expected Result	API returns HTTP 200 and a structured response containing the requested settings for the company.
Actual Result	API returns HTTP 200 and a structured response containing the requested settings for the company.
Status	Pass
Severity	Medium

File: Admin\CompanyTest.txt

Test ID	TC-Adm-CT-001
Title	Verify User Creation via Form Request in Company Creation API
Objective	Ensure the CompaniesController::store action uses CompaniesRequest for validation of request data, confirming integration between controller and form request.
Preconditions	- Application is running with all dependencies. - Database is seeded with initial and demo data. - API is authenticated via Sanctum, acting as an admin user.
Steps	Step 1: Initiate a check to see if CompaniesController::store uses CompaniesRequest. Step 2: Evaluate assertion that validates use of the form request.
Test Data	- No actual data submitted; verifies route configuration. - Expected: CompaniesController::store method must require CompaniesRequest.
Expected Result	CompaniesController::store action uses CompaniesRequest, ensuring all company-creation submissions are validated by this form request.
Actual Result	CompaniesController::store action uses CompaniesRequest, as validated.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-002
Title	Validate Company Creation via API with Database Persistence
Objective	Ensure POST /api/v1/companies creates a new company with specified attributes and persists it to the database.
Preconditions	- Application is running. - Database is seeded with initial and demo data. - Authenticated as an admin user via Sanctum.
Steps	Step 1: Send POST request to /api/v1/companies with company payload. Step 2: Assert HTTP status 201 (created). Step 3: Verify the database contains the company record with the provided name.
Test Data	- Input: Company payload with fields: name (from factory), currency = 12, address: country_id = 12. - Expected: Company is created in DB with given name.
Expected Result	API returns status 201; a new company exists in 'companies' table with specified name from payload.
Actual Result	API returns status 201; company is persisted in database as expected.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-003
Title	Validate Error Response for Incorrect Company Deletion Request
Objective	Verify API returns validation error when attempting to delete a company with invalid input format.
Preconditions	- Application is running. - Database is seeded with initial and demo data. - Authenticated as an admin user via Sanctum.
Steps	Step 1: Send POST request to /api/v1/companies/delete with invalid payload. Step 2: Assert that the API responds with HTTP status 422.

Test Data	- Input: Request body ["xyz"] (invalid format, not valid company IDs). - Expected: API rejects the request and responds with status 422.
Expected Result	API responds with HTTP status 422 indicating a validation error due to invalid company deletion input.
Actual Result	API responds with HTTP status 422 as expected for invalid deletion input.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CT-004
Title	Validate Ownership Transfer to Another User for a Company
Objective	Ensure that company ownership can be successfully transferred to another user via the API.
Preconditions	- Application is running. - Database is seeded with initial and demo data. - Authenticated as an admin user via Sanctum. - At least one company and one additional user exist.
Steps	Step 1: Create a company record. Step 2: Create a user to transfer ownership to. Step 3: Send POST request to ownership transfer endpoint using the target user ID. Step 4: Assert the API responds with HTTP 200 (OK).
Test Data	- Input: Target user ID (from factory-created user). - Endpoint: /api/v1/transfer/ownership/{user_id} - Expected: API responds with success (HTTP 200).
Expected Result	API responds with HTTP 200, indicating successful ownership transfer to the specified user.
Actual Result	API responds with HTTP 200, and ownership transfer is successful.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-005
Title	Verify Company List Retrieval via API
Objective	Ensure that the API successfully retrieves a list of existing companies.
Preconditions	- Application is running. - Database is seeded with initial and demo data. - Authenticated as an admin user via Sanctum.
Steps	Step 1: Send GET request to /api/v1/companies. Step 2: Assert the API response is HTTP 200 (OK).
Test Data	- Input: None (GET request to /api/v1/companies). - Expected: API returns HTTP 200 and a list of companies.
Expected Result	API responds with HTTP 200 and returns the list of companies.
Actual Result	API responds with HTTP 200, companies are listed as expected.
Status	Pass
Severity	High

File: Admin\ConfigTest.txt

Test ID	TC-Adm-CT-001
Title	Retrieve All Supported Languages via Config API
Objective	Validate that the API endpoint returns the complete list of supported languages, ensuring integration between the configuration controller, related database seeders, and API authentication.
Preconditions	Application running and accessible via HTTP Database seeded with DatabaseSeeder and DemoSeeder Authenticated user (Sanctum API token) with a company header ID
Steps	Step 1: Trigger GET request with key=languages and proper headers Step 2: Validate HTTP status is 200 OK Step 3: Confirm JSON response structure contains language data as per seeders
Test Data	Input: GET request to api/v1/config?key=languages Expected Output: HTTP 200 OK, JSON containing list of supported languages (as seeded in DB)
Expected Result	API returns HTTP 200 OK and a JSON array of all supported languages, matching the seeded configuration.
Actual Result	API returns HTTP 200 OK and a JSON array of all supported languages, matching the seeded configuration.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-002
Title	Retrieve All Fiscal Years via Config API
Objective	Verify that the configuration API provides the list of available fiscal years, confirming integration between controller logic, database content, and API access roles.
Preconditions	Application is operational Database seeded with fiscal year data (DatabaseSeeder/DemoSeeder) Authenticated API user with company context
Steps	Step 1: Send GET request with key=fiscal_years and necessary headers Step 2: Check that response status is 200 OK Step 3: Review JSON body for presence of fiscal year data
Test Data	Input: GET request to api/v1/config?key=fiscal_years Expected Output: HTTP 200 OK, JSON containing all fiscal year options
Expected Result	API returns HTTP 200 and a JSON object/array enumerating all fiscal years seeded in the database.
Actual Result	API returns HTTP 200 and a JSON object/array enumerating all fiscal years seeded in the database.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-003
Title	Fetch All Estimate Conversion Options via Config API
Objective	Ensure the estimate conversion options are available through the configuration API, demonstrating successful data flow from seeders to endpoint.
Preconditions	Application instance up and running Database seeded with estimate options Authenticated user session, with company ID in headers

Steps	Step 1: Execute GET request to endpoint with key=convert_estimate_options Step 2: Validate response status is 200 OK Step 3: Inspect JSON for correct estimate conversion options
Test Data	Input: GET request to api/v1/config?key=convert_estimate_options Expected Output: HTTP 200 OK, JSON array of conversion options
Expected Result	Response contains HTTP 200 with a complete JSON array of estimate conversion options as defined in seeders.
Actual Result	Response contains HTTP 200 with a complete JSON array of estimate conversion options as defined in seeders.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CT-004
Title	Obtain All Retrospective Edit Options via Config API
Objective	Check the integration between controller, database, and API authentication by retrieving all retrospective edit options from the configuration endpoint.
Preconditions	Application accessible through HTTP API Database seeded with retrospective edit configurations Authenticated user with company header
Steps	Step 1: Make GET request with key=retrospective_edits Step 2: Confirm HTTP response code is 200 OK Step 3: Verify JSON body includes correct retrospective edit options
Test Data	Input: GET request to api/v1/config?key=retrospective_edits Expected Output: HTTP 200 OK, JSON data of retrospective edit options
Expected Result	API returns HTTP 200 OK and JSON array of all retrospective edit options as configured.
Actual Result	API returns HTTP 200 OK and JSON array of all retrospective edit options as configured.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CT-005
Title	List All Currency Converter Servers via Config API
Objective	Validate that the list of available currency converter server integrations is correctly returned via the API endpoint, ensuring system-level configuration is exposed to clients.
Preconditions	Application running Seeded database with currency converter server configurations API user authenticated via Sanctum
Steps	Step 1: Issue GET request with key=currency_converter_servers and required headers Step 2: Verify HTTP 200 OK response Step 3: Check JSON for complete and correct server data
Test Data	Input: GET request to api/v1/config?key=currency_converter_servers Expected Output: HTTP 200 OK, JSON array of server options
Expected Result	API responds with HTTP 200, including a JSON array listing all currency converter servers from the DB seeders.
Actual Result	API responds with HTTP 200, including a JSON array listing all currency converter servers from the DB seeders.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CT-006
Title	Retrieve All Exchange Rate Drivers via Config API
Objective	Ensure the system exposes all supported exchange rate driver configurations via the API and that they are available as seeded.
Preconditions	Application and HTTP API operational Seeded database with exchange rate driver details User authenticated with company context
Steps	Step 1: Send GET request for key=exchange_rate_drivers Step 2: Confirm HTTP 200 status Step 3: Validate that JSON response matches seeded driver configurations
Test Data	Input: GET request to api/v1/config?key=exchange_rate_drivers Expected Output: 200 OK with JSON array of drivers
Expected Result	API returns HTTP 200 and JSON data listing all exchange rate drivers present in the seeders.
Actual Result	API returns HTTP 200 and JSON data listing all exchange rate drivers present in the seeders.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CT-007
Title	Fetch All Custom Field Models via Config API
Objective	Verify integration between database seeders and API layer by checking that custom field models are returned from the config API endpoint.
Preconditions	Application instance running Custom field models present in seeded database Authenticated user context with company header
Steps	Step 1: Make API call with key=custom_field_models Step 2: Check response for HTTP 200 OK Step 3: Inspect JSON for accurate custom field model list
Test Data	Input: GET request to api/v1/config?key=custom_field_models Expected Output: HTTP 200 OK, JSON array of custom field models
Expected Result	API responds with HTTP 200 OK, and a JSON array of all custom field models as defined in the data seeders.
Actual Result	API responds with HTTP 200 OK, and a JSON array of all custom field models as defined in the data seeders.
Status	Pass
Severity	Medium

File: Admin\CurrenciesTest.txt

Test ID	TC-Adm-CT-001
Title	Retrieve the List of Used Currencies via API
Objective	Validate that the integration between the API and database correctly provides a list of all currencies currently used in the system for the authenticated user.
Preconditions	- Application must be running and accessible. - Database must be seeded with initial and demo data using DatabaseSeeder and DemoSeeder. - API endpoint /api/v1/currencies/used must be accessible. - User authentication must be performed using Laravel Sanctum, with appropriate company headers set.
Steps	Step 1: Seed the database with required data using DatabaseSeeder and DemoSeeder. Step 2: Authenticate as user ID 1 and set company header. Step 3: Send GET request to /api/v1/currencies/used. Step 4: Verify that the response status is HTTP 200 (OK).
Test Data	- Authenticated user with ID 1. - Company header set to the first company associated with user 1. - No request body; GET request to endpoint /api/v1/currencies/used. - Expected HTTP status: 200 OK.
Expected Result	The API responds with a 200 OK status, confirming that the user's used currencies are successfully retrieved from the integrated database and authentication layers.
Actual Result	The API responds with a 200 OK status, confirming that the user's used currencies are successfully retrieved from the integrated database and authentication layers.
Status	Pass
Severity	Medium

File: Admin\CustomFieldTest.txt

Test ID	TC-Adm-CFT-001
Title	Retrieve list of custom fields via API
Objective	Validate that the API endpoint correctly returns a list of custom fields and is accessible with proper authentication and company context.
Preconditions	- Application is running with API server accessible. - Database seeded using DatabaseSeeder and DemoSeeder. - Authenticated as a valid user with valid company context in headers.
Steps	Step 1: Send GET request to api/v1/custom-fields?page=1 with proper authentication and company headers. Step 2: Verify that the response status is HTTP 200 OK.
Test Data	- GET request to endpoint: api/v1/custom-fields?page=1
Expected Result	API responds with HTTP 200 and returns a paginated JSON list of custom fields belonging to the authenticated user's company.
Actual Result	API responds with HTTP 200 and returns a paginated JSON list of custom fields belonging to the authenticated user's company.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CFT-002
Title	Create a new custom field via API
Objective	Verify that the API endpoint successfully creates a new custom field, and that the database is updated accordingly.
Preconditions	- Application is running with API server accessible. - Database seeded using DatabaseSeeder and DemoSeeder. - Authenticated as a valid user with valid company context in headers.
Steps	Step 1: Send POST request to api/v1/custom-fields with generated custom field payload. Step 2: Verify that the response status is HTTP 201 Created. Step 3: Check the database for the presence of the new custom field with the submitted attributes.
Test Data	- POST request to endpoint: api/v1/custom-fields - Payload: Custom field data (name, label, type, model_type, is_required) from a factory Expected database values: - name, label, type, model_type, is_required (matching the payload)
Expected Result	API responds with HTTP 201 Created, and the custom field is present in the custom_fields database table with the submitted values.
Actual Result	API responds with HTTP 201 Created, and the custom field is present in the custom_fields database table with the submitted values.
Status	Pass
Severity	High

Test ID	TC-Adm-CFT-003
Title	Ensure store endpoint uses CustomFieldRequest for validation
Objective	Verify that the CustomFieldsController::store method integrates and requires the CustomFieldRequest for validating incoming requests, ensuring business rules are consistently enforced.
Preconditions	- Application code for CustomFieldsController and CustomFieldRequest is present. - Database seeded using DatabaseSeeder and DemoSeeder.

Steps	Step 1: Assert that the store method on CustomFieldsController uses CustomFieldRequest for input validation.
Test Data	- Method: CustomFieldsController::store - Form Request: CustomFieldRequest
Expected Result	CustomFieldsController::store method invokes CustomFieldRequest for validation, ensuring all incoming data is validated as per business rules before storing.
Actual Result	CustomFieldsController::store method invokes CustomFieldRequest for validation, ensuring all incoming data is validated as per business rules before storing.
Status	Pass
Severity	High

Test ID	TC-Adm-CFT-004
Title	Update an existing custom field via API
Objective	Confirm that the API endpoint processes updates for an existing custom field and that changes are correctly reflected in the database.
Preconditions	- Application is running with API server accessible. - Database seeded using DatabaseSeeder and DemoSeeder. - At least one custom field exists in the database. - Authenticated as a valid user with valid company context in headers.
Steps	Step 1: Create a custom field in the database for updating. Step 2: Send PUT request to api/v1/custom-fields/{customFieldId} with updated custom field payload. Step 3: Verify that the response status is HTTP 200 OK. Step 4: Check the database for updated values for the custom field.
Test Data	- PUT request to endpoint: api/v1/custom-fields/{customFieldId} - Payload: Updated custom field data (including is_required: false) - Existing custom field: generated by factory Expected database values: - id, name, label, type, model_type (matching updated payload)
Expected Result	API responds with HTTP 200 OK, and the custom field’s database record is updated with the new values (name, label, type, model_type, is_required).
Actual Result	API responds with HTTP 200 OK, and the custom field’s database record is updated with the new values (name, label, type, model_type, is_required).
Status	Pass
Severity	High

Test ID	TC-Adm-CFT-005
Title	Ensure update endpoint uses CustomFieldRequest for validation
Objective	Verify that the CustomFieldsController::update method integrates and requires the CustomFieldRequest for validating incoming data, ensuring business logic consistency.
Preconditions	- Application code for CustomFieldsController and CustomFieldRequest is present. - Database seeded using DatabaseSeeder and DemoSeeder.
Steps	Step 1: Assert that the update method on CustomFieldsController uses CustomFieldRequest for validating update requests.
Test Data	- Method: CustomFieldsController::update - Form Request: CustomFieldRequest
Expected Result	CustomFieldsController::update method invokes CustomFieldRequest for input validation, ensuring all update data is validated according to business rules.

Actual Result	CustomFieldsController::update method invokes CustomFieldRequest for input validation, ensuring all update data is validated according to business rules.
Status	Pass
Severity	High

Test ID	TC-Adm-CFT-006
Title	Delete a custom field via API
Objective	Ensure that the API endpoint successfully deletes a custom field and the deletion is reflected in the database.
Preconditions	- Application is running with API server accessible. - Database seeded using DatabaseSeeder and DemoSeeder. - At least one custom field exists in the database. - Authenticated as a valid user with valid company context in headers.
Steps	Step 1: Ensure a custom field exists by creating one with the factory. Step 2: Send DELETE request to api/v1/custom-fields/{customFieldId}. Step 3: Verify HTTP response status is 200 OK and JSON response includes { "success": true }. Step 4: Confirm the custom field is deleted from the database.
Test Data	- DELETE request to endpoint: api/v1/custom-fields/{customFieldId} - Existing custom field: generated by factory Expected JSON response: - { "success": true } Expected database effect: - Custom field is deleted from database
Expected Result	API responds with HTTP 200 OK and JSON { "success": true }, and the specified custom field is removed from the custom_fields table.
Actual Result	API responds with HTTP 200 OK and JSON { "success": true }, and the specified custom field is removed from the custom_fields table.
Status	Pass
Severity	High

File: Admin\CustomerTest.txt

Test ID	TC-Adm-CT-001
Title	Retrieve Paginated List of Customers via API
Objective	Validate that the customer listing API endpoint returns a paginated set of customers from the database and is accessible to authenticated users.
Preconditions	- Application is running. - Database is seeded with customer data and necessary demo information. - An authenticated user with necessary permissions is present. - API is accessible.
Steps	Step 1: Send GET request to 'api/v1/customers?page=1' as an authenticated user. Step 2: Verify that the response status is 200 OK.
Test Data	- Input: GET request to 'api/v1/customers?page=1' - Expected values: HTTP 200 OK response.
Expected Result	API returns a paginated customer list with HTTP status 200 OK, confirming accessibility and data retrieval.
Actual Result	API returns a paginated customer list with HTTP status 200 OK, confirming accessibility and data retrieval.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-002
Title	Retrieve Customer Statistics via API
Objective	Verify that the API endpoint returns statistics for a given customer, with correct integration between customer and invoice data.
Preconditions	- Application is running. - Database seeded with customer and invoice data. - Authenticated user session established. - API accessible.
Steps	Step 1: Create a customer in the database. Step 2: Create an invoice linked to this customer. Step 3: Send GET request to 'api/v1/customers/{customerId}/stats'. Step 4: Verify response status is 200.
Test Data	- Input: GET request to 'api/v1/customers/{customerId}/stats' - Expected values: HTTP 200 OK; customer with at least one invoice.
Expected Result	API returns statistics for the customer, confirming correct interaction between customer and invoice modules with HTTP status 200.
Actual Result	API returns statistics for the customer, confirming correct interaction between customer and invoice modules with HTTP status 200.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CT-003
Title	Create New Customer via API and Validate Database Insertion
Objective	Ensure that the create customer endpoint correctly inserts a new customer into the database and handles shipping/billing information.
Preconditions	- Application running. - Database seeded and accessible. - API accessible; user authenticated.

Steps	Step 1: Prepare customer payload containing shipping and billing data. Step 2: Send POST request to 'api/v1/customers'. Step 3: Verify HTTP 200 OK response. Step 4: Assert that the database contains a new customer record with specified name and email.
Test Data	- Input: POST request to 'api/v1/customers' with fields: - shipping: { name: 'newName', address_street_1: 'address' } - billing: { name: 'newName', address_street_1: 'address' } - Expected values: Customer record with provided name and email is present in the database; HTTP 200 OK.
Expected Result	API responds with 200 OK and the new customer is successfully created in the database, with correct billing and shipping details.
Actual Result	API responds with 200 OK and the new customer is successfully created in the database, with correct billing and shipping details.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-004
Title	Validate Customer Creation Enforces Form Request Validation
Objective	Confirm that customer creation uses the defined CustomerRequest validation, ensuring all business rules are checked before storing data.
Preconditions	- Application running. - Relevant form request (CustomerRequest) configured. - API accessible; user authenticated.
Steps	Step 1: Retrieve CustomersController 'store' action. Step 2: Assert that 'store' action uses CustomerRequest for validation.
Test Data	- Input: API endpoint 'store' action. - Expected values: Validation is enforced at the controller level using CustomerRequest.
Expected Result	The controller’s store action enforces the defined form request validation, protecting business logic integrity.
Actual Result	The controller’s store action enforces the defined form request validation, protecting business logic integrity.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-005
Title	Retrieve Single Customer Details via API and Validate Database
Objective	Ensure that individual customer information returned by the API matches the database record.
Preconditions	- Application running. - Database seeded with customer data. - API accessible; user authenticated.
Steps	Step 1: Create a customer record in the database. Step 2: Send GET request to 'api/v1/customers/{customerId}'. Step 3: Assert the database contains the correct customer info. Step 4: Verify HTTP 200 OK response.
Test Data	- Input: GET request to 'api/v1/customers/{customerId}'. Customer exists in DB. - Expected values: HTTP 200 OK; DB record for customer id, name, email.
Expected Result	API returns the correct customer information with HTTP 200 OK, and the database contains matching customer data.
Actual Result	API returns the correct customer information with HTTP 200 OK, and the database contains matching customer data.

Status	Pass
Severity	High

Test ID	TC-Adm-CT-006
Title	Update Customer Information via API and Validate Database Changes
Objective	Verify that updates to a customer's billing and shipping information via the API are reflected correctly in the database.
Preconditions	- Application running. - Database seeded with customer data. - API accessible; user authenticated.
Steps	Step 1: Create a customer record in the database. Step 2: Construct update payload with new shipping and billing data. Step 3: Send PUT request to 'api/v1/customers/{customerId}' with payload. Step 4: Assemble expected update fields, including 'email' and 'creator_id'. Step 5: Assert HTTP 200 OK response. Step 6: Check that the customer record in DB matches updated information.
Test Data	- Input: PUT request to 'api/v1/customers/{customerId}' with new billing and shipping info. - Expected values: Updated customer record in DB, HTTP 200 OK.
Expected Result	API responds with 200 OK, and the database reflects the updated customer information with correct billing, shipping, email, and creator_id.
Actual Result	API responds with 200 OK, and the database reflects the updated customer information with correct billing, shipping, email, and creator_id.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-007
Title	Validate Customer Update Enforces Form Request Validation
Objective	Ensure that customer update requests use CustomerRequest validation, enforcing all rules for updating customer data.
Preconditions	- Application running. - Relevant form request (CustomerRequest) configured. - API accessible; user authenticated.
Steps	Step 1: Retrieve CustomersController 'update' action. Step 2: Assert that 'update' action uses CustomerRequest for validation.
Test Data	- Input: API endpoint 'update' action. - Expected values: Validation is enforced at the controller level using CustomerRequest.
Expected Result	Controller’s update action enforces form request validation, ensuring business rules are respected when modifying customer data.
Actual Result	Controller’s update action enforces form request validation, ensuring business rules are respected when modifying customer data.
Status	Pass
Severity	High

Test ID	TC-Adm-CT-008
Title	Search Customers via API with Specific Filters
Objective	Validate that the customer list endpoint supports searching and filtering by parameters such as name and email address.
Preconditions	- Application running. - Database contains multiple customer records. - API accessible; user authenticated.

Steps	Step 1: Define filter parameters for search (name/email). Step 2: Build request query string from filters. Step 3: Send GET request to 'api/v1/customers' with query string. Step 4: Assert HTTP 200 OK response.
Test Data	- Input: GET request to 'api/v1/customers?' with filters: - page: 1, limit: 15, search: 'doe', email: '.com' - Expected values: HTTP 200 OK; filtered customer results.
Expected Result	API responds with 200 OK and returns customers filtered by the provided search and email parameters.
Actual Result	API responds with 200 OK and returns customers filtered by the provided search and email parameters.
Status	Pass
Severity	Medium

Test ID	TC-Adm-CT-009
Title	Delete Multiple Customers via API and Validate Batch Deletion
Objective	Ensure that the API allows batch deletion of customers and returns confirmation of success.
Preconditions	- Application running. - Database seeded with multiple customers. - API accessible; user authenticated.
Steps	Step 1: Create multiple customer records in the database. Step 2: Collect IDs for customers to be deleted. Step 3: Send POST request with 'ids' array to 'api/v1/customers/delete'. Step 4: Assert response is HTTP 200 OK and JSON contains 'success': true.
Test Data	- Input: POST request to 'api/v1/customers/delete' with 'ids' array containing customer IDs. - Expected values: HTTP 200 OK; JSON response { success: true }.
Expected Result	API responds with 200 OK and a JSON object indicating success; specified customers are deleted from the database.
Actual Result	API responds with 200 OK and a JSON object indicating success; specified customers are deleted from the database.
Status	Pass
Severity	High

File: Admin\DashboardTest.txt

Test ID	TC-Adm-DT-001
Title	Verify Dashboard API Accessibility and Basic Response
Objective	Validate the integrated behavior between the authentication system, company context header, and the Dashboard API endpoint to ensure authorized users can successfully access the dashboard data.
Preconditions	- Application server running and accessible - Database seeded with initial core and demo data (DatabaseSeeder, DemoSeeder) - User exists with ID 1 and is associated with at least one company - User is authenticated via Laravel Sanctum - Appropriate API headers (company context) are set for the request
Steps	Step 1: Seed database with required data using DatabaseSeeder and DemoSeeder Step 2: Authenticate as User ID 1 and set company header for request context Step 3: Send GET request to api/v1/dashboard as the authenticated user Step 4: Validate that the response status is HTTP 200 OK
Test Data	- Authenticated user: ID 1 - Company header: ID of first company linked to user - Request: GET api/v1/dashboard - Expected HTTP Status: 200 OK
Expected Result	API responds successfully with HTTP 200 OK, confirming authorized user can access dashboard endpoint and required authentication, company context, and data integration are functioning.
Actual Result	API responds successfully with HTTP 200 OK, confirming authorized user can access dashboard endpoint and required authentication, company context, and data integration are functioning.
Status	Pass
Severity	High

Test ID	TC-Adm-DT-002
Title	Validate Dashboard Search API Accessible and Returns Success for Name Filter
Objective	Confirm that an authenticated user with a valid company context can access the Dashboard Search API, filter results using a 'name' query parameter, and receive a successful response, verifying integration between API, authentication, and DB search functionality.
Preconditions	- Application server active and API endpoints exposed - Database fully seeded including searchable demo data - User with ID 1 exists and is linked to company - User is authenticated and company context header is set
Steps	Step 1: Seed database via DatabaseSeeder and DemoSeeder for complete search data Step 2: Authenticate as User ID 1 and add company header for API context Step 3: Issue GET request to api/v1/search?name=ab to invoke name-filtered search Step 4: Assert response status is HTTP 200 OK, confirming API, authentication, and database integration
Test Data	- Authenticated user: ID 1 - Company header: First company ID for user - Request: GET api/v1/search?name=ab - Query parameter: name=ab - Expected HTTP Status: 200 OK
Expected Result	Dashboard Search API returns HTTP 200 OK for name-filtered request, demonstrating endpoint is integrated and accessible for authenticated users with seeded data.

Actual Result	Dashboard Search API returns HTTP 200 OK for name-filtered request, demonstrating endpoint is integrated and accessible for authenticated users with seeded data.
Status	Pass
Severity	Medium

File: Admin\EstimateTest.txt

Test ID	TC-Adm-ET-001
Title	Retrieve Paginated List of Estimates via API
Objective	Validate that the estimates API endpoint correctly returns a paginated list of estimates for authenticated users.
Preconditions	- Application is running. - Database is seeded with sample estimates using DatabaseSeeder and DemoSeeder. - API is accessible and user is authenticated via Sanctum.
Steps	1. Send a GET request to the /estimates endpoint with page query parameter. 2. Validate HTTP 200 OK response.
Test Data	- GET request to api/v1/estimates?page=1. - No payload required.
Expected Result	API returns status 200 OK with a paginated list of estimates.
Actual Result	API returns status 200 OK with a paginated list of estimates.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-002
Title	Create New Estimate with Items and Taxes
Objective	Confirm interaction between API, database, and business logic for successful creation of an estimate with associated items and taxes.
Preconditions	- Application is running. - Database is seeded. - API and DB are accessible. - User authenticated via Sanctum.
Steps	1. Prepare estimate data with required fields and related items/taxes. 2. Send POST request to /estimates endpoint with payload. 3. Validate response status 201 Created. 4. Ensure the created estimate and related entities exist in the database.
Test Data	- POST request to api/v1/estimates. - Payload includes estimate_number, items array, and taxes array. Expected DB values: All core estimate attributes match the request data.
Expected Result	API returns 201 Created; new estimate, items, and taxes are persisted with matching details.
Actual Result	API returns 201 Created; new estimate, items, and taxes are persisted with matching details.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-003
Title	Form Request Validation for Estimate Creation
Objective	Verify that the estimate creation endpoint uses correct form request validation for input sanitization and authorization.
Preconditions	- Application running. - EstimatesController available.
Steps	1. Assert that EstimatesController@store uses EstimatesRequest form request.
Test Data	- None (test asserts controller wiring).
Expected Result	Estimate creation action is validated using EstimatesRequest.

Actual Result	Estimate creation action is validated using EstimatesRequest.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-004
Title	Update Existing Estimate and Associated Items/Taxes
Objective	Ensure that updates to an existing estimate, including its items and taxes, are correctly handled and changes are persisted in the database.
Preconditions	- Application running. - Database seeded with existing estimate. - API and DB accessible; user authenticated.
Steps	1. Prepare initial estimate with items and taxes. 2. Prepare updated estimate data. 3. Send PUT request to /estimates/{id} with payload. 4. Validate database for updated estimate fields and item/tax relationships. 5. Check HTTP 200 OK response.
Test Data	- PUT request to api/v1/estimates/{estimate_id} with updated estimate data (including items/taxes). - DB should reflect updated fields.
Expected Result	Estimate, associated items, and taxes are updated; response is 200 OK.
Actual Result	Estimate, associated items, and taxes are updated; response is 200 OK.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-005
Title	Form Request Validation for Estimate Update
Objective	Ensure update action on estimates uses EstimatesRequest for proper input validation and authorization.
Preconditions	- Application running. - EstimatesController available.
Steps	1. Assert that EstimatesController@update uses EstimatesRequest.
Test Data	- None (asserts controller wiring).
Expected Result	Estimate update action is validated with EstimatesRequest.
Actual Result	Estimate update action is validated with EstimatesRequest.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-006
Title	Search and Filter Estimates via API
Objective	Test the API's ability to filter, search, and paginate estimates according to provided criteria.
Preconditions	- Application running. - Database seeded with sample estimates. - API accessible and user authenticated.
Steps	1. Send GET request to endpoint with filters. 2. Validate HTTP 200 response.
Test Data	- GET request to api/v1/estimates with filters: page=1, limit=15, search=doe, from_date=2020-07-18, to_date=2020-07-20, estimate_number=000003.
Expected Result	API returns 200 OK and a filtered list of estimates according to query parameters.

Actual Result	API returns 200 OK and a filtered list of estimates according to query parameters.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-007
Title	Form Request Validation for Sending Estimates
Objective	Ensure that the send estimate endpoint uses correct form request validation for email sending.
Preconditions	- Application running. - SendEstimateController available.
Steps	1. Assert SendEstimateController@__invoke uses SendEstimatesRequest.
Test Data	- None (asserts controller wiring).
Expected Result	Send estimate action is validated with SendEstimatesRequest.
Actual Result	Send estimate action is validated with SendEstimatesRequest.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-008
Title	Send Estimate to Customer via Email API
Objective	Validate integrated behavior for emailing an estimate to the customer, ensuring API, email subsystem, and DB interact properly.
Preconditions	- Application running. - Database seeded with an estimate. - Mail system faked for test isolation. - User authenticated.
Steps	1. Create estimate in DB. 2. Send POST request to /estimates/{id}/send with email details. 3. Assert 200 OK and {success: true} in response. 4. Assert SendEstimateMail was triggered.
Test Data	- POST to api/v1/estimates/{estimate_id}/send with: subject: test body: test from: john@example.com to: doe@example.com
Expected Result	API returns 200 OK and triggers SendEstimateMail for the correct customer; response includes success.
Actual Result	API returns 200 OK and triggers SendEstimateMail for the correct customer; response includes success.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-009
Title	Mark Estimate as Accepted via API
Objective	Ensure the API can mark an estimate's status as accepted, updating DB and responding to client correctly.
Preconditions	- Application running. - Database seeded. - API accessible; user authenticated.

Steps	1. Create an estimate. 2. Send POST request to /estimates/{id}/status with status. 3. Assert 200 OK and {success: true} in response. 4. Confirm status in DB is updated to accepted.
Test Data	- POST to api/v1/estimates/{estimate_id}/status with status=Estimate::STATUS_ACCEPTED
Expected Result	API returns 200 OK with success; DB status of estimate is set to accepted.
Actual Result	API returns 200 OK with success; DB status of estimate is set to accepted.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-010
Title	Mark Estimate as Rejected via API
Objective	Verify API ability to set an estimate's status to rejected and persist in database.
Preconditions	- Application running. - Database seeded. - API accessible; user authenticated.
Steps	1. Create an estimate. 2. Send POST request to /estimates/{id}/status with status. 3. Assert 200 OK and {success: true} in response. 4. Verify status in DB is updated to rejected.
Test Data	- POST to api/v1/estimates/{estimate_id}/status with status=Estimate::STATUS_REJECTED
Expected Result	API returns 200 OK with success; DB status of estimate is set to rejected.
Actual Result	API returns 200 OK with success; DB status of estimate is set to rejected.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-011
Title	Convert Estimate to Invoice via API
Objective	Validate end-to-end conversion of an estimate to an invoice, ensuring correct API, business logic, and persistence.
Preconditions	- Application running. - Database seeded. - API accessible; user authenticated.
Steps	1. Create an estimate. 2. Send POST to /estimates/{id}/convert-to-invoice. 3. Verify HTTP 200 response.
Test Data	- POST to api/v1/estimates/{estimate_id}/convert-to-invoice
Expected Result	API returns 200 OK; estimate is successfully converted to an invoice.
Actual Result	API returns 200 OK; estimate is successfully converted to an invoice.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-012
Title	Form Request Validation for Bulk Estimate Deletion
Objective	Confirm that deletion of multiple estimates via API uses correct form request validation for secure bulk actions.

Preconditions	- Application running. - EstimatesController available.
Steps	1. Assert EstimatesController@delete uses DeleteEstimatesRequest form request.
Test Data	- None (asserts controller wiring).
Expected Result	Bulk delete action is validated with DeleteEstimatesRequest.
Actual Result	Bulk delete action is validated with DeleteEstimatesRequest.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-013
Title	Delete Multiple Estimates via API
Objective	Verify batch deletion of multiple estimates via API, confirming DB changes and API response.
Preconditions	- Application running. - Database seeded with batch of estimates. - API accessible; user authenticated.
Steps	1. Create several estimates. 2. Send POST request with array of estimate IDs to delete. 3. Verify HTTP 200 and {success: true} in response. 4. Assert each estimate is deleted in database.
Test Data	- POST to api/v1/estimates/delete with ids of estimates to delete.
Expected Result	API returns 200 OK; selected estimates are deleted from database and success is reported.
Actual Result	API returns 200 OK; selected estimates are deleted from database and success is reported.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-014
Title	Retrieve All Estimate Templates
Objective	Confirm API endpoint retrieves available estimate templates for customization without error.
Preconditions	- Application running. - Templates exist in DB. - API accessible; user authenticated.
Steps	1. Send GET request to endpoint for estimate templates. 2. Assert HTTP 200 response.
Test Data	- GET request to api/v1/estimates/templates
Expected Result	API returns status 200 OK with list of available estimate templates.
Actual Result	API returns status 200 OK with list of available estimate templates.
Status	Pass
Severity	Low

Test ID	TC-Adm-ET-015
Title	Create Estimate with Tax Per Item
Objective	Validate the creation of an estimate where each item has its own tax, including proper database relationships and calculations.

Preconditions	- Application running. - Database seeded. - API and DB accessible; user authenticated.
Steps	1. Construct estimate data with multiple items, each having its own tax. 2. Send POST request to /estimates endpoint. 3. Assert status 201 Created. 4. Verify database contains estimate, items, and associated taxes.
Test Data	- POST to api/v1/estimates - estimate_number, tax_per_item = YES - Multiple items, each with its own taxes
Expected Result	Estimate with tax per item is created; all items and taxes are correctly persisted in the database; response is 201.
Actual Result	Estimate with tax per item is created; all items and taxes are correctly persisted in the database; response is 201.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-016
Title	Create Estimate with EUR Currency and Exchange Rates
Objective	Verify creation of an estimate using EUR currency, including exchange rates and base amounts for each item and tax.
Preconditions	- Application running. - Database and currency/exchange rates populated. - API accessible; user authenticated.
Steps	1. Prepare estimate data reflecting EUR currency and exchange rate calculations. 2. Send POST to /estimates endpoint. 3. Assert status 201 Created. 4. Validate all EUR and exchange-rate related fields in DB for estimates, items, and taxes.
Test Data	- POST to api/v1/estimates - All relevant fields: discount_type, exchange_rate (86.403538), base_* fields, item/tax details.
Expected Result	Estimate in EUR created with all exchange-rate fields correctly persisted; all related items and taxes recorded; response is 201.
Actual Result	Estimate in EUR created with all exchange-rate fields correctly persisted; all related items and taxes recorded; response is 201.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-017
Title	Update Existing Estimate with EUR Currency and Exchange Rates
Objective	Confirm updating of an existing estimate including currency (EUR), exchange rates, and all related fields and relationships.
Preconditions	- Application running. - Database seeded with initial estimate in default currency. - API accessible; user authenticated.
Steps	1. Create initial estimate with referenced items/taxes. 2. Prepare updated payload with EUR currency and exchange rate related fields. 3. Send PUT request to /estimates/{id}. 4. Assert HTTP 200 OK. 5. Validate all updated fields and relationships in DB.

Test Data	- PUT to api/v1/estimates/{estimate_id} - All relevant EUR/exchange rate fields in payload: base amounts, exchange_rate, updated items/taxes.
Expected Result	Selected estimate, items, and taxes updated to reflect EUR and exchange rate calculations; response is 200 OK.
Actual Result	Selected estimate, items, and taxes updated to reflect EUR and exchange rate calculations; response is 200 OK.
Status	Pass
Severity	High

File: Admin\ExpenseCategoryTest.txt

Test ID	TC-Adm-ECT-001
Title	Retrieve All Expense Categories via API
Objective	Validate that the API endpoint correctly retrieves the list of all expense categories, verifying integration between API controller and database.
Preconditions	Application running in testing environment. Database seeded with test expense category data using DatabaseSeeder and DemoSeeder. Authenticated user is present and associated company ID is set in request headers.
Steps	Step 1: Send GET request to 'api/v1/categories' endpoint. Step 2: Verify that the response returns HTTP 200 OK status.
Test Data	No explicit input data (GET request to 'api/v1/categories'). Expected result: HTTP 200 OK status with a JSON array of categories.
Expected Result	API returns HTTP 200 OK; JSON response contains the list of all expense categories available for the authenticated user's company.
Actual Result	API returns HTTP 200 OK; JSON response contains the list of all expense categories available for the authenticated user's company.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ECT-002
Title	Create a New Expense Category via API
Objective	Validate that the API endpoint allows creation of a new expense category and stores it properly in the database, checking integration between API, DB, and form submission.
Preconditions	Application running in testing environment. Database seeded with baseline data; no conflicting categories. Authenticated user is present with company ID set.
Steps	Step 1: Send POST request to 'api/v1/categories' with generated expense category payload. Step 2: Verify API response status is 201 Created. Step 3: Validate that the new category exists in 'expense_categories' table with correct fields.
Test Data	Input: A valid raw expense category payload (name, description) generated via factory. Expected values: HTTP 201 status; database entry with matching 'name' and 'description'.
Expected Result	HTTP 201 response with new category created; corresponding row exists in expense_categories table matching input values.
Actual Result	HTTP 201 response with new category created; corresponding row exists in expense_categories table matching input values.
Status	Pass
Severity	High

Test ID	TC-Adm-ECT-003
Title	Expense Category Creation Validates Input via Form Request
Objective	Verify that the category creation endpoint in the controller uses a Form Request for validation, ensuring robust input validation is integrated at the controller boundary.
Preconditions	Application running; environment set for PHPUnit/Pest integration. Controllers and ExpenseCategoryRequest implemented.

Steps	Step 1: Assert that 'store' action of controller uses ExpenseCategoryRequest for validating input.
Test Data	No explicit test data (tests internal integration with request validation mechanism). Expected value: Controller method 'store' uses ExpenseCategoryRequest.
Expected Result	ExpenseCategoriesController::store method is mapped with ExpenseCategoryRequest for validating incoming category creation data.
Actual Result	ExpenseCategoriesController::store method is mapped with ExpenseCategoryRequest for validating incoming category creation data.
Status	Pass
Severity	High

Test ID	TC-Adm-ECT-004
Title	Retrieve Specific Expense Category Details via API
Objective	Ensure the API endpoint correctly fetches details for a specific expense category, validating interaction between API layer and database record retrieval.
Preconditions	Application running with test database seeded. At least one expense category exists in the database. Authenticated user present; correct company header set.
Steps	Step 1: Create an expense category in the database. Step 2: Send GET request to 'api/v1/categories/{id}' using created category's ID. Step 3: Assert that API returns HTTP 200 OK.
Test Data	Input: ID of a created ExpenseCategory. Expected value: HTTP 200 OK status code.
Expected Result	API returns HTTP 200 OK with details for the requested expense category in the JSON response.
Actual Result	API returns HTTP 200 OK with details for the requested expense category in the JSON response.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ECT-005
Title	Update Existing Expense Category via API
Objective	Validate that the API correctly updates an existing expense category and persists changes in the database, testing proper integration between API PUT endpoint and the DB.
Preconditions	Application running in test environment. Database seeded; at least one expense category exists. Authenticated user present; company ID specified.
Steps	Step 1: Create an expense category in the database. Step 2: Prepare new category data via factory. Step 3: Send PUT request to 'api/v1/categories/{id}' with updated data. Step 4: Assert API response is HTTP 200 OK. Step 5: Verify that the category's record in the database matches updated name and description.
Test Data	Input: Existing category ID; new expense category payload (name, description). Expected value: HTTP 200 OK; category updated in database with new values.
Expected Result	Expense category is updated in database; API returns HTTP 200 OK; database row reflects new name and description for given ID.

Actual Result	Expense category is updated in database; API returns HTTP 200 OK; database row reflects new name and description for given ID.
Status	Pass
Severity	High

Test ID	TC-Adm-ECT-006
Title	Expense Category Update Validates Input via Form Request
Objective	Ensure that the update endpoint of the expense category controller uses the correct Form Request for input validation, verifying controller and validation layer integration.
Preconditions	Application running; Pest/PHPUnit test environment. Controllers and ExpenseCategoryRequest are available.
Steps	Step 1: Assert that ExpenseCategoriesController::update method uses ExpenseCategoryRequest for validating update input.
Test Data	No explicit input data (asserts application of validation logic). Expected value: Controller method 'update' uses ExpenseCategoryRequest.
Expected Result	ExpenseCategoriesController::update method is integrated with ExpenseCategoryRequest validation for update operations.
Actual Result	ExpenseCategoriesController::update method is integrated with ExpenseCategoryRequest validation for update operations.
Status	Pass
Severity	High

Test ID	TC-Adm-ECT-007
Title	Delete Expense Category via API
Objective	Validate that the API correctly deletes a specified expense category and confirms removal from the database, ensuring proper integration between DELETE endpoint and database operations.
Preconditions	Application running in testing environment. Database seeded with at least one expense category. Authenticated user present; company ID header set.
Steps	Step 1: Create an expense category. Step 2: Send DELETE request to 'api/v1/categories/{id}'. Step 3: Assert response is HTTP 200 OK and JSON contains { 'success': true }. Step 4: Verify that category has been deleted from database.
Test Data	Input: ID of existing expense category. Expected values: HTTP 200 OK; JSON response { 'success': true }; category no longer exists in database.
Expected Result	Expense category is removed from database; API returns HTTP 200 OK with { 'success': true }.
Actual Result	Expense category is removed from database; API returns HTTP 200 OK with { 'success': true }.
Status	Pass
Severity	High

File: Admin\ExpenseTest.txt

Test ID	TC-Adm-ET-001
Title	Retrieve Expense List Successfully
Objective	Validate that the expense list API correctly returns expenses paginated for an authenticated user.
Preconditions	- Application running with required services (API & DB). - Database seeded with expense and user data. - User authentication via Sanctum.
Steps	1. Send GET request to api/v1/expenses?page=1 as authenticated user. 2. Check HTTP response code.
Test Data	- GET request to endpoint: api/v1/expenses?page=1 - Authenticated user header with company ID.
Expected Result	API returns HTTP 200 status with array of expense objects matching pagination for the user.
Actual Result	API returns HTTP 200 status with paginated list of expense objects.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-002
Title	Create Expense and Persist to Database
Objective	Verify that submitting a valid expense via API results in creation and persistence in the database.
Preconditions	- Application running with API and database. - Database seeded; at least one user and expense category present. - User authenticated.
Steps	1. Prepare expense data (factory + custom values). 2. Send POST request to api/v1/expenses with expense data. 3. Validate HTTP response status is 201 (created). 4. Confirm expense record exists in expenses table with all payload attributes.
Test Data	- POST payload: expense with amount=150, exchange_rate=76.217498, base_amount=11432.6247, and generated fields. - Endpoint: api/v1/expenses
Expected Result	API responds with HTTP 201 status and expense record exists in database with all given values.
Actual Result	API responds with HTTP 201 status and expense data is correctly persisted.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-003
Title	Validate Expense Store Endpoint Uses Form Request
Objective	Ensure the expense creation endpoint enforces request validation by using the correct form request class.
Preconditions	- Application bootstrapped; routes loaded. - ExpensesController accessible.
Steps	1. Assert that ExpensesController@store uses ExpenseRequest for validation.
Test Data	- Reference to ExpenseRequest class.
Expected Result	ExpensesController's store action utilizes ExpenseRequest validation for incoming data.
Actual Result	ExpensesController@store employs ExpenseRequest for data validation.

Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-004
Title	Retrieve Single Expense Successfully
Objective	Validate retrieval of an individual expense via API, including DB existence and field matching.
Preconditions	- Application running. - Database seeded; expense category and user present. - Expense instance created. - User authenticated.
Steps	1. Create expense in database with set date. 2. Send GET request for expense by ID. 3. Assert HTTP response is 200 OK. 4. Confirm expense record matches in DB by ID and main fields.
Test Data	- GET request to: api/v1/expenses/{expense_id} - Created expense with known attributes ('expense_date' => '2019-02-05').
Expected Result	API responds with HTTP 200 and correct expense data; database contains matching expense.
Actual Result	API returns 200 status with expense in DB as expected.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-005
Title	Update Expense and Verify Changes
Objective	Check that update API correctly modifies expense attributes in the database for an authenticated user.
Preconditions	- Application and DB running. - Expense exists in DB. - User authenticated.
Steps	1. Create an expense in database. 2. Prepare updated expense attributes. 3. Send PUT request to update expense by ID. 4. Confirm 200 OK response. 5. Verify expense record has updated attributes in DB.
Test Data	- PUT request to: api/v1/expenses/{expense_id} - New expense data (factory-generated).
Expected Result	API responds with 200 OK; expense record in DB updated with new data.
Actual Result	API returns 200 OK; expense record matches new data.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-006
Title	Validate Expense Update Endpoint Uses Form Request
Objective	Ensure expense update API enforces validation by using the ExpenseRequest form request.
Preconditions	- Application running with routes. - ExpensesController available.
Steps	1. Assert ExpensesController@update utilizes ExpenseRequest for input validation.
Test Data	- Reference to ExpenseRequest class for validation.

Expected Result	ExpensesController's update action uses ExpenseRequest for validating update data.
Actual Result	ExpenseRequest used for update validation.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-007
Title	Search Expenses Using Filters
Objective	Confirm that the expense listing API supports filtering by category, search string, and date range and returns valid results.
Preconditions	- Application running; DB seeded. - Expense data present for given filters. - User authenticated.
Steps	1. Prepare query string with filters. 2. Send GET request to expenses endpoint with filters. 3. Validate HTTP 200 OK response.
Test Data	- Filters: page=1, limit=15, expense_category_id=1, search='cate', from_date='2020-07-18', to_date='2020-07-20' - GET endpoint: api/v1/expenses?{query_string}
Expected Result	API returns 200 status with expense list matching filter criteria.
Actual Result	API responds with 200 OK, filtered expense data included.
Status	Pass
Severity	Medium

Test ID	TC-Adm-ET-008
Title	Delete Multiple Expenses in Batch Operation
Objective	Verify that the batch delete endpoint removes all specified expenses and returns success.
Preconditions	- Application running; DB seeded. - Multiple expenses exist in DB. - Valid user authentication.
Steps	1. Create three expenses in database. 2. Send POST request to deletion endpoint with IDs. 3. Assert response is 200 OK and 'success' field is true. 4. Check each expense is deleted from the database.
Test Data	- POST payload: {ids: [expense_id1, expense_id2, expense_id3]} - Endpoint: api/v1/expenses/delete
Expected Result	API responds with 200 OK, success=true, and all specified expenses are deleted from database.
Actual Result	API indicates success and all expenses are deleted as expected.
Status	Pass
Severity	High

Test ID	TC-Adm-ET-009
Title	Update Expense with EUR Currency and Verify Persistence
Objective	Validate update API when editing an expense to use EUR currency, correct numeric fields, and ensure DB persistence.
Preconditions	- Application & database running; currencies supported. - Expense created in DB. - Authenticated user.

Steps	1. Create known expense. 2. Prepare EUR expense update data. 3. Send PUT request to update expense by ID. 4. Assert 200 OK response. 5. Validate database reflects updated expense values.
Test Data	- Expense ID for update. - New expense data: amount=150, exchange_rate=76.217498, base_amount=11432.6247 - PUT endpoint: api/v1/expenses/{expense_id}
Expected Result	API responds with 200 OK and the expense is updated in database with correct EUR currency data.
Actual Result	API responds as expected and database shows accurate updated values.
Status	Pass
Severity	Medium

File: Admin\FileDiskTest.txt

Test ID	TC-Adm-FDT-001
Title	Verify Retrieval of All File Disks via API
Objective	Ensure API integration retrieves the list of all file disks from the database and returns a successful response.
Preconditions	- Application server running with all dependencies installed. - Database seeded with FileDisk and User data (via DatabaseSeeder and DemoSeeder). - User authenticated using Sanctum; valid company header set.
Steps	Step 1: Send a GET request to /api/v1/disks as authenticated user. Step 2: Assert that the response status is 200 OK. Step 3: (Implied) Validate that response contains the list of disks.
Test Data	- No payload; authenticated GET request to /api/v1/disks. - Expected: HTTP 200 OK response, and a JSON array of file disks.
Expected Result	The API returns HTTP 200 OK and a JSON array containing the file disk data from the database.
Actual Result	The API returns HTTP 200 OK and a JSON array containing the file disk data from the database.
Status	Pass
Severity	High

Test ID	TC-Adm-FDT-002
Title	Validate Creation of a New File Disk via API
Objective	Confirm that a new file disk can be created through the API, and integration successfully persists it into the database.
Preconditions	- Application server up; all dependencies installed. - Database seeded; User authenticated via Sanctum; company header present. - FileDisk factory available for generating test data.
Steps	Step 1: Generate raw file disk data using factory. Step 2: POST the generated data to /api/v1/disks. Step 3: Format credentials as JSON and assert the new disk exists in the file_disks table.
Test Data	- Input: Raw file disk data (e.g., driver, credentials, etc.) generated via factory. - Expected: HTTP 200 OK response; database contains new disk including credentials encoded as JSON.
Expected Result	A new file disk is created, API returns HTTP 200 OK, and the disk, including its credentials (JSON-encoded), is present in the database.
Actual Result	A new file disk is created, API returns HTTP 200 OK, and the disk, including its credentials (JSON-encoded), is present in the database.
Status	Pass
Severity	High

Test ID	TC-Adm-FDT-003
Title	Verify Update of Existing File Disk via API
Objective	Ensure that updating an existing file disk via API interaction correctly alters the record in the database.
Preconditions	- Application server running; all dependencies installed. - Database seeded; authenticated user in context; company header set. - Existing FileDisk available (created via factory).

Steps	Step 1: Create an initial file disk using factory and persist to database. Step 2: Generate new raw disk data for update. Step 3: Send PUT request to /api/v1/disks/{disk_id} with updated data. Step 4: Encode credentials as JSON and assert presence of updated disk data in database.
Test Data	- Input: Existing file disk object; New raw file disk data for the update. - Expected: HTTP 200 OK response after PUT; database record matches updated data (with credentials JSON-encoded).
Expected Result	The file disk is updated in the database with new values, credentials are JSON-encoded, and the API returns HTTP 200 OK.
Actual Result	The file disk is updated in the database with new values, credentials are JSON-encoded, and the API returns HTTP 200 OK.
Status	Pass
Severity	High

Test ID	TC-Adm-FDT-004
Title	Validate Retrieval of Specific File Disk via Driver Identifier
Objective	Confirm API integration correctly retrieves details for a specific file disk using its driver identifier.
Preconditions	- Backend application running; dependencies installed. - Database properly seeded; authenticated user active; company header set. - FileDisk instance created and available.
Steps	Step 1: Create a file disk in the database using factory. Step 2: Send GET request to /api/v1/disks/{disk.driver}. Step 3: Assert response returns status 200 and includes expected disk data.
Test Data	- Input: FileDisk object (created via factory); GET request to /api/v1/disks/{driver}. - Expected: HTTP 200 OK response with file disk details.
Expected Result	The API returns HTTP 200 OK and a JSON object with the requested file disk's details.
Actual Result	The API returns HTTP 200 OK and a JSON object with the requested file disk's details.
Status	Pass
Severity	Medium

Test ID	TC-Adm-FDT-005
Title	Verify Retrieval of Available File Disk Drivers via API
Objective	Ensure the API provides a list of all available file disk drivers integrated into the application.
Preconditions	- Application server running and all routes loaded. - Database seeded; authenticated user context set; company header present.
Steps	Step 1: Send a GET request to /api/v1/disk/drivers as an authenticated user. Step 2: Assert response status is 200 OK. Step 3: (Implied) Check that response contains a list of driver identifiers.
Test Data	- No input payload; GET request to /api/v1/disk/drivers. - Expected: HTTP 200 OK response; JSON array of driver identifiers.
Expected Result	API returns HTTP 200 OK and a JSON array containing available file disk driver identifiers.
Actual Result	API returns HTTP 200 OK and a JSON array containing available file disk driver identifiers.
Status	Pass
Severity	Low

File: Admin\InvoiceTest.txt

Test ID	TC-Adm-IT-001
Title	Retrieve overdue invoices via API
Objective	Validate integration between the invoice API and the database to retrieve overdue invoices and ensure API filtering and pagination works correctly.
Preconditions	Application running; database seeded with invoices. API endpoint /api/v1/invoices accessible. User authenticated.
Steps	Step 1: Send GET request to /api/v1/invoices?page=1&type;=OVERDUE&limit;=20. Step 2: Verify HTTP status is 200 OK.
Test Data	GET request with query parameters: page=1, type=OVERDUE, limit=20.
Expected Result	The API responds with HTTP 200 OK and returns a paginated list of overdue invoices matching the criteria.
Actual Result	The API responds with HTTP 200 OK and returns a paginated list of overdue invoices matching the criteria.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-002
Title	Create invoice via API and persist items and taxes in database
Objective	Validate that creating an invoice via API stores invoice metadata, items, and taxes correctly in the database.
Preconditions	Application running; database seeded. Tax and InvoiceItem factories available. API endpoint /api/v1/invoices accessible. User authenticated.
Steps	Step 1: Send POST request to /api/v1/invoices with complete invoice payload. Step 2: Verify HTTP status is 200 OK. Step 3: Validate invoice and item are stored in DB with correct data.
Test Data	POST payload with invoice details, one tax, and one item. Expected invoice fields: template_name, invoice_number, sub_total, discount, customer_id, total, tax. Expected item fields: item_id, name.
Expected Result	Invoice, item, and tax entries are persisted in the database matching the input. API responds with HTTP 200 OK.
Actual Result	Invoice, item, and tax entries are persisted in the database matching the input. API responds with HTTP 200 OK.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-003
Title	Create invoice marked as sent and validate persistence
Objective	Ensure that invoice creation via API correctly marks the invoice as 'sent' and persists all fields, items, and taxes.
Preconditions	Application running; database seeded. Tax and InvoiceItem factories available. API endpoint /api/v1/invoices accessible. User authenticated.
Steps	Step 1: Send POST request to /api/v1/invoices with invoice payload. Step 2: Verify HTTP status is 200 OK. Step 3: Confirm invoice and related item data exists in the database.

Test Data	POST payload with invoice details, marked as sent, including relevant fields and one tax/item each.
Expected Result	Invoice and item are saved in the database with all sent-related fields set and HTTP 200 OK response.
Actual Result	Invoice and item are saved in the database with all sent-related fields set and HTTP 200 OK response.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-004
Title	Validate store API enforces form request validation
Objective	Ensure the invoice 'store' action uses the designated form request for payload validation, confirming integrated validation control.
Preconditions	Application running. InvoicesController::store action implemented. InvoicesRequest exists.
Steps	Step 1: Assert that InvoicesController@store uses InvoicesRequest for validation.
Test Data	No input data; verification is against controller setup and request handling.
Expected Result	InvoicesController@store validates input using InvoicesRequest form request.
Actual Result	InvoicesController@store validates input using InvoicesRequest form request.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-005
Title	Update invoice via API and verify changes in database
Objective	Ensure the API updates invoice fields, items, and taxes, and changes are reflected in the database.
Preconditions	Application running. Invoice exists in database. API endpoint /api/v1/invoices/{id} accessible. User authenticated.
Steps	Step 1: Create existing invoice. Step 2: Send PUT request to update invoice with new data. Step 3: Verify HTTP status is 200 OK. Step 4: Assert updated invoice/item/tax exist in DB.
Test Data	PUT payload with new invoice data, one tax, and one item. Existing invoice with id, invoice_date, due_date.
Expected Result	Invoice, items, and taxes are updated in DB to match new data; API responds with HTTP 200 OK.
Actual Result	Invoice, items, and taxes are updated in DB to match new data; API responds with HTTP 200 OK.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-006
Title	Validate update API enforces form request validation
Objective	Verify the 'update' endpoint for invoices uses the required form request for validating input, ensuring proper controller integration.
Preconditions	Application running. InvoicesController::update with mapped InvoicesRequest form.

Steps	Step 1: Assert InvoicesController@update uses InvoicesRequest.
Test Data	No input data; verification of controller-configured validation.
Expected Result	InvoicesController@update action validates using InvoicesRequest.
Actual Result	InvoicesController@update action validates using InvoicesRequest.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-007
Title	Send invoice to customer and verify email notification and status update
Objective	Ensure sending invoice triggers email delivery, updates invoice status, and API response signals success.
Preconditions	Application running; database seeded with invoice. Mail system is faked for test. API endpoint /api/v1/invoices/{id}/send accessible.
Steps	Step 1: Create invoice in DB. Step 2: Send POST request to /send endpoint with email details. Step 3: Assert HTTP 200 OK and success==true JSON. Step 4: Verify invoice status is set to SENT. Step 5: Confirm that SendInvoiceMail is triggered.
Test Data	POST payload: from, to, subject, body. Invoice with id.
Expected Result	Invoice status changes to SENT and email is sent; success==true is returned; HTTP 200 OK.
Actual Result	Invoice status changes to SENT and email is sent; success==true is returned; HTTP 200 OK.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-008
Title	Mark invoice as paid via API and validate status in DB
Objective	Verify marking an invoice as paid via the API updates both invoice and paid_status fields in the database.
Preconditions	Application running; invoice exists. API endpoint /api/v1/invoices/{id}/status accessible.
Steps	Step 1: Create test invoice. Step 2: Send status update via POST. Step 3: Assert HTTP 200 OK and success==true JSON. Step 4: Check invoice paid_status==STATUS_PAID in DB.
Test Data	POST payload: status=STATUS_COMPLETED. Invoice with status.
Expected Result	Invoice's paid_status becomes STATUS_PAID; API returns success==true.
Actual Result	Invoice's paid_status becomes STATUS_PAID; API returns success==true.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-009
Title	Mark invoice as sent via API and validate status in DB
Objective	Ensure marking an invoice as sent through the API updates invoice status in the database and returns correct response.

Preconditions	Application running; invoice exists. API endpoint /api/v1/invoices/{id}/status accessible.
Steps	Step 1: Create test invoice. Step 2: Send POST request to update status. Step 3: Verify HTTP 200 OK and success==true JSON. Step 4: Validate invoice status is STATUS_SENT in DB.
Test Data	POST payload: status=STATUS_SENT. Invoice with status.
Expected Result	Invoice status set to STATUS_SENT; API returns success==true.
Actual Result	Invoice status set to STATUS_SENT; API returns success==true.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-010
Title	Search invoices with filters via API
Objective	Verify that search endpoint respects filter parameters and returns correct results.
Preconditions	Application running. Invoices exist in DB. API endpoint /api/v1/invoices accessible.
Steps	Step 1: Build query string with filters. Step 2: Send GET request to /api/v1/invoices. Step 3: Assert HTTP 200 OK response.
Test Data	GET parameters: page=1, limit=15, search='doe', status=STATUS_DRAFT, from_date=2019-01-20, to_date=2019-01-27, invoice_number='000012'.
Expected Result	API successfully returns filtered invoice data, HTTP 200 OK.
Actual Result	API successfully returns filtered invoice data, HTTP 200 OK.
Status	Pass
Severity	Medium

Test ID	TC-Adm-IT-011
Title	Batch delete multiple invoices via API and validate database removal
Objective	Ensure API can delete multiple invoices and the database reflects their removal.
Preconditions	Application running; multiple invoices exist. API endpoint /api/v1/invoices/delete accessible.
Steps	Step 1: Create 3 invoices. Step 2: Send batch delete POST request. Step 3: Assert HTTP 200 OK and success==true JSON. Step 4: Confirm deleted invoices are absent in DB.
Test Data	POST payload: ids = [list of invoice IDs].
Expected Result	All specified invoices are deleted from DB; API returns success==true.
Actual Result	All specified invoices are deleted from DB; API returns success==true.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-012
Title	Clone invoice via API and verify creation of new invoice
Objective	Test invoice cloning functionality via API and database, ensuring new invoice is created with status 201.

Preconditions	Application running; invoice exists. API endpoint /api/v1/invoices/{id}/clone accessible.
Steps	Step 1: Create original invoice. Step 2: Send POST request to /clone endpoint. Step 3: Assert HTTP 201 Created.
Test Data	Invoice with id to be cloned.
Expected Result	API creates a cloned invoice and responds with HTTP 201.
Actual Result	API creates a cloned invoice and responds with HTTP 201.
Status	Pass
Severity	Medium

Test ID	TC-Adm-IT-013
Title	Create invoice with negative tax value via API and persist in DB
Objective	Ensure system accepts and stores invoices with negative tax values, verifying tax, item, and invoice records in DB.
Preconditions	Application running. Tax, Invoice, InvoiceItem factories operative. API endpoint /api/v1/invoices accessible.
Steps	Step 1: POST invoice data with negative tax. Step 2: Assert HTTP 200 OK. Step 3: Confirm invoice, item, tax records in DB.
Test Data	POST payload: invoice data with tax percent=-9.99, one item.
Expected Result	Invoice, item, and tax entries (with negative tax) are persisted in DB; API returns HTTP 200 OK.
Actual Result	Invoice, item, and tax entries (with negative tax) are persisted in DB; API returns HTTP 200 OK.
Status	Pass
Severity	Medium

Test ID	TC-Adm-IT-014
Title	Create invoice with per-item taxation via API and verify persistence
Objective	Ensure invoices with per-item taxes are stored correctly and tax records reflect per-item tax.
Preconditions	Application running. API endpoint /api/v1/invoices accessible. Factories for Invoice, InvoiceItem, Tax available.
Steps	Step 1: Send POST request with detailed payload. Step 2: Assert HTTP 200 OK. Step 3: Validate invoice, items, and related tax records in DB.
Test Data	POST payload: invoice with tax_per_item='YES', two items each with tax.
Expected Result	Invoice and all item taxes are stored in DB; API returns HTTP 200 OK.
Actual Result	Invoice and all item taxes are stored in DB; API returns HTTP 200 OK.
Status	Pass
Severity	Medium

Test ID	TC-Adm-IT-015
Title	Create invoice with EUR currency and complex financial fields
Objective	Validate that invoices with EUR currency and related exchange rates, calculations and taxes are stored as expected.

Preconditions	Application running. API endpoint /api/v1/invoices accessible. Factories available for Invoice, InvoiceItem, Tax.
Steps	Step 1: Send POST request with EUR-currency invoice data. Step 2: Assert HTTP 200 OK. Step 3: Validate invoice, item, and tax with correct currency and calculations in DB.
Test Data	POST payload: invoice with EUR currency, exchange_rate, base_* and derived fields, one tax, one item with translation fields.
Expected Result	Invoice, item, and tax with EUR currency and all respective fields are stored in DB; API returns HTTP 200 OK.
Actual Result	Invoice, item, and tax with EUR currency and all respective fields are stored in DB; API returns HTTP 200 OK.
Status	Pass
Severity	Medium

Test ID	TC-Adm-IT-016
Title	Update invoice with EUR currency and verify complex field persistence
Objective	Ensure updating an invoice with EUR currency modifies all related exchange rate and calculation fields as expected in the database.
Preconditions	Application running; invoice with EUR currency exists. API endpoint /api/v1/invoices/{id} accessible.
Steps	Step 1: Create invoice with EUR currency. Step 2: Send PUT request to update with new EUR and calculated fields. Step 3: Assert HTTP 200 OK. Step 4: Validate invoice, item, and tax data with correct fields in DB.
Test Data	PUT payload: all EUR currency fields, including exchange_rate, base_total, taxes, items.
Expected Result	Invoice, item, and tax fields are updated in DB to reflect all provided EUR currency fields and calculations; API returns HTTP 200 OK.
Actual Result	Invoice, item, and tax fields are updated in DB to reflect all provided EUR currency fields and calculations; API returns HTTP 200 OK.
Status	Pass
Severity	Medium

File: Admin\ItemTest.txt

Test ID	TC-Adm-IT-001
Title	Retrieve paginated list of items via API
Objective	Validate that the integration between the Items API endpoint and the database correctly returns a paginated list of items for an authenticated user.
Preconditions	Application running with Laravel and database connected Database seeded via DatabaseSeeder and DemoSeeder User with ID 1 exists and is authenticated using Sanctum Company header set for requests
Steps	Step 1: Send GET request to api/v1/items?page=1 Step 2: Verify that the response status is HTTP 200 OK
Test Data	Input: GET api/v1/items?page=1 Expected: HTTP 200 OK response with paginated list of items belonging to the user's company
Expected Result	The API returns HTTP 200 OK and responds with a paginated list of items for the authenticated user's company.
Actual Result	The API returns HTTP 200 OK and responds with a paginated list of items for the authenticated user's company.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-002
Title	Create a new item with associated taxes via API
Objective	Validate that the Items API endpoint creates a new item along with its related taxes, and persists them in the database as expected.
Preconditions	Application running; database seeded User with ID 1 authenticated; company header set Tax and Item models exist
Steps	Step 1: Prepare item data (name, description, price, company_id, taxes array) Step 2: Send POST request to api/v1/items with the item data Step 3: Assert that the item exists in the 'items' table with provided details Step 4: Assert that the associated taxes exist in the 'taxes' table with correct item_id Step 5: Verify response status is HTTP 200 OK
Test Data	Input: POST api/v1/items, payload containing item fields and an array of two tax objects Expected: HTTP 200 OK; item row created with provided details; associated taxes recorded with correct item_id
Expected Result	Item and associated taxes are created in the database, and API responds with HTTP 200 OK including new item data.
Actual Result	Item and associated taxes are created in the database, and API responds with HTTP 200 OK including new item data.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-003
Title	Verify that item creation uses proper form request validation
Objective	Ensure that the ItemsController@store action in the API integrates ItemsRequest form validation at the controller entrypoint.
Preconditions	Application running ItemsController and ItemsRequest classes present

Steps	Step 1: Check that ItemsController@store uses ItemsRequest for validation Step 2: Assert integration between controller and form request exists
Test Data	Input: None (reflective integration check) Expected: ItemsController@store is typehinted to use ItemsRequest for validating creation requests
Expected Result	ItemsController@store action uses ItemsRequest for request validation, ensuring integrated validation logic.
Actual Result	ItemsController@store action uses ItemsRequest for request validation, ensuring integrated validation logic.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-004
Title	Retrieve a single item by ID via API
Objective	Validate integration between API endpoint and database, ensuring that retrieving an item by its ID returns correct item data for the authorized user.
Preconditions	Application running; database seeded Item exists in database for user company User is authenticated; company header set
Steps	Step 1: Create a test item in the database Step 2: Send GET request to api/v1/items/{item_id} Step 3: Verify HTTP 200 OK response Step 4: Assert that item with expected details exists in 'items' table
Test Data	Input: GET api/v1/items/{item_id} Expected: HTTP 200 OK; JSON response contains correct item attributes
Expected Result	API returns HTTP 200 OK and the JSON response for the specified item, matching the details in the database.
Actual Result	API returns HTTP 200 OK and the JSON response for the specified item, matching the details in the database.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-005
Title	Update an existing item and its taxes via API
Objective	Validate that the API endpoint updates an item’s details and associated taxes in the database, confirming end-to-end update functionality.
Preconditions	Application running; database seeded User authenticated; company header set Item exists in 'items' table
Steps	Step 1: Create an item in the database Step 2: Prepare updated item data (fields and single tax) Step 3: Send PUT request to api/v1/items/{item_id} with updated data Step 4: Verify response status is HTTP 200 OK Step 5: Assert that updated item is present in 'items' table with new values Step 6: Assert that tax entry is linked to updated item in 'taxes' table
Test Data	Input: PUT api/v1/items/{item_id}, payload with updated item fields and single tax object Expected: HTTP 200 OK; updated item and taxes exist in the database with new values
Expected Result	API responds with HTTP 200 OK, item updated in 'items', taxes updated/linked in 'taxes', reflecting provided changes.
Actual Result	API responds with HTTP 200 OK, item updated in 'items', taxes updated/linked in 'taxes', reflecting provided changes.

Status	Pass
Severity	High

Test ID	TC-Adm-IT-006
Title	Verify that item update uses proper form request validation
Objective	Ensure that the ItemsController@update action integrates ItemsRequest validation to correctly validate update requests at the controller layer.
Preconditions	Application running ItemsController and ItemsRequest classes available
Steps	Step 1: Check that ItemsController@update uses ItemsRequest for request validation Step 2: Assert integration of controller and ItemsRequest exists
Test Data	Input: None (structural validation check) Expected: ItemsController@update uses ItemsRequest for validation
Expected Result	ItemsController@update action uses ItemsRequest to validate incoming update requests.
Actual Result	ItemsController@update action uses ItemsRequest to validate incoming update requests.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-007
Title	Bulk deletion of multiple items via API
Objective	Confirm that the API endpoint can delete multiple items in one request, and that deleted items are removed from the database.
Preconditions	Application running; database seeded User authenticated; company header set At least five items exist in the database
Steps	Step 1: Create five items in the database Step 2: Prepare array of their IDs Step 3: Send POST request to api/v1/items/delete with the ID array Step 4: Verify HTTP 200 OK response Step 5: Assert that each item is deleted from the database
Test Data	Input: POST api/v1/items/delete, payload: {ids: [array of five item IDs]} Expected: HTTP 200 OK; item rows deleted from the database
Expected Result	All specified items are deleted from the database in a single request, and API responds with HTTP 200 OK.
Actual Result	All specified items are deleted from the database in a single request, and API responds with HTTP 200 OK.
Status	Pass
Severity	High

Test ID	TC-Adm-IT-008
Title	Search items using multiple filters via API
Objective	Ensure that the Items API endpoint correctly integrates search, paging, and filtering logic, returning response data according to query filters.
Preconditions	Application running; database seeded User authenticated; company header set Items exist matching filters (name, price, unit)

Steps	Step 1: Prepare filters for search: page=1, limit=15, search='doe', price=6, unit='kg' Step 2: Send GET request with these query parameters to api/v1/items Step 3: Verify response status is HTTP 200 OK
Test Data	Input: GET api/v1/items?page=1&limit;=15&search;=doe&price;=6&unit;=kg Expected: HTTP 200 OK; Items in response match the provided search terms and filters
Expected Result	API returns HTTP 200 OK with list of items matching all provided search and filter options.
Actual Result	API returns HTTP 200 OK with list of items matching all provided search and filter options.
Status	Pass
Severity	Medium

File: Admin\LocationTest.txt

Test ID	TC-Adm-LT-001
Title	Validate retrieval of country data via API
Objective	Ensure that the API endpoint for fetching the list of countries returns a successful response and properly integrates with the seeding/database layer.
Preconditions	Application running in a test-capable environment with Laravel framework. Database seeded with both DatabaseSeeder and DemoSeeder to ensure country data is present. User authentication configured via Sanctum; valid user set as authenticated. HTTP headers set to include valid company ID from seeded user.
Steps	Step 1: Perform GET request to 'api/v1/countries' with valid headers and authentication. Step 2: Receive JSON response from API. Step 3: Assert that response status is HTTP 200 OK indicating successful retrieval.
Test Data	API request: GET to 'api/v1/countries' Authenticated user with ID 1 associated with company No additional input parameters Expected value: HTTP 200 OK response code
Expected Result	API returns HTTP 200 OK status for countries endpoint, confirming country data is retrievable and the integration between API authentication, seeded database, and routing is successful.
Actual Result	API returns HTTP 200 OK status for countries endpoint, confirming country data is retrievable and the integration between API authentication, seeded database, and routing is successful.
Status	Pass
Severity	Medium

File: Admin\NextNumberTest.txt

Test ID	TC-Adm-NNT-001
Title	Validate Next Number Generation for Financial Document Keys via API
Objective	Verify that the API endpoint correctly generates and returns the next sequential number for different financial document types (invoice, estimate, payment), ensuring proper integration between the API logic and database numbering scheme.
Preconditions	- Application server is running with all required services active - Database seeded with essential tables and data (via DatabaseSeeder and DemoSeeder) - API is accessible and authentication is setup using an existing valid user (User ID 1) - HTTP request headers are set to include a valid company context
Steps	Step 1: Seed the database with all required baseline and demonstration data. Step 2: Establish an authenticated session as User #1 via Sanctum and set the HTTP company header. Step 3: Send a GET request to api/v1/next-number with key=invoice; verify status 200 and nextNumber as 'INV-000001' in JSON response. Step 4: Send a GET request to api/v1/next-number with key=estimate; verify status 200 and nextNumber as 'EST-000001' in JSON response. Step 5: Send a GET request to api/v1/next-number with key=payment; verify status 200 and nextNumber as 'PAY-000001' in JSON response.
Test Data	Inputs: - GET request to api/v1/next-number?key=invoice - GET request to api/v1/next-number?key=estimate - GET request to api/v1/next-number?key=payment Expected Values: - For "invoice": Response JSON includes { 'nextNumber': 'INV-000001' } - For "estimate": Response JSON includes { 'nextNumber': 'EST-000001' } - For "payment": Response JSON includes { 'nextNumber': 'PAY-000001' }
Expected Result	Each GET request to the next-number API with document-type keys returns HTTP 200 and a JSON response containing the correct next number prefixed for the respective document type ('INV-000001', 'EST-000001', 'PAY-000001'), demonstrating that the numbering system and API are correctly integrated and initialized.
Actual Result	Each GET request to the next-number API with document-type keys returns HTTP 200 and a JSON response containing the correct next number prefixed for the respective document type ('INV-000001', 'EST-000001', 'PAY-000001'), demonstrating that the numbering system and API are correctly integrated and initialized.
Status	Pass
Severity	High

File: Admin\NotesTest.txt

Test ID	TC-Adm-NT-001
Title	Retrieve All Notes via API
Objective	Verify that the notes retrieval endpoint returns a correct list of notes for an authenticated user and ensures API and DB integration.
Preconditions	- Application server running and accessible. - Database seeded with required data via DatabaseSeeder and DemoSeeder. - User ID 1 exists and is authenticated with Sanctum. - At least one note exists in the database.
Steps	Step 1: Send GET request to /api/v1/notes with required headers. Step 2: Verify HTTP 200 OK status in response.
Test Data	- HTTP GET request to /api/v1/notes. - Authenticated headers for user and company. Expected values: - HTTP status 200. - Response contains a list of note objects.
Expected Result	The API responds with HTTP 200 and a JSON containing all notes accessible to the authenticated user.
Actual Result	The API responds with HTTP 200 and a JSON containing all notes accessible to the authenticated user.
Status	Pass
Severity	Medium

Test ID	TC-Adm-NT-002
Title	Create a New Note via API and Verify Database Entry
Objective	Ensure the API allows creating a new note and that the note is persisted in the database correctly.
Preconditions	- Application server running. - Database seeded using DatabaseSeeder and DemoSeeder. - User ID 1 authenticated with Sanctum.
Steps	Step 1: Generate a new note data object using the factory. Step 2: Send POST request to /api/v1/notes with generated note data. Step 3: Assert response status is 201 Created. Step 4: Check that the note exists in the 'notes' table in the database with the provided data.
Test Data	- Note creation data generated via Note::factory()->raw(). - HTTP POST request to /api/v1/notes with note data. Expected values: - HTTP status 201. - Note exists in database with provided values.
Expected Result	The API responds with HTTP 201 Created and the 'notes' table contains the new note matching the provided data.
Actual Result	The API responds with HTTP 201 Created and the 'notes' table contains the new note matching the provided data.
Status	Pass
Severity	High

Test ID	TC-Adm-NT-003
Title	Retrieve Specific Note via API
Objective	Confirm that an individual note can be queried by ID and retrieved from the database via API.

Preconditions	- Application server running. - Database seeded. - User ID 1 authenticated with Sanctum. - At least one note created for the test.
Steps	Step 1: Create a note record in the database using the factory. Step 2: Send GET request to /api/v1/notes/{note->id} with authentication headers. Step 3: Assert response status is 200 OK.
Test Data	- Note instance created via Note::factory()->create(). - HTTP GET request to /api/v1/notes/{id}, where {id} is the test note's ID. Expected values: - HTTP status 200. - Response contains the details of the requested note.
Expected Result	The API responds with HTTP 200 OK and the JSON contains the specific note data as stored in the database.
Actual Result	The API responds with HTTP 200 OK and the JSON contains the specific note data as stored in the database.
Status	Pass
Severity	Medium

Test ID	TC-Adm-NT-004
Title	Update Existing Note via API and Verify Persistence
Objective	Validate that the API allows updating an existing note, and that the database values reflect the new data.
Preconditions	- Application server running. - Database seeded. - User ID 1 authenticated with Sanctum. - At least one note created for this test.
Steps	Step 1: Create a note in the database. Step 2: Prepare updated note data via factory. Step 3: Send PUT request to /api/v1/notes/{note->id} with the updated data. Step 4: Assert response status is 200 OK. Step 5: Check that updated note data exists in the 'notes' table in the database.
Test Data	- Existing note created via Note::factory()->create(). - Updated note data via Note::factory()->raw(). - HTTP PUT request to /api/v1/notes/{id} with updated data. Expected values: - HTTP status 200. - Updated note data present in database.
Expected Result	The API responds with HTTP 200 OK and the database contains the updated note data after the request.
Actual Result	The API responds with HTTP 200 OK and the database contains the updated note data after the request.
Status	Pass
Severity	High

Test ID	TC-Adm-NT-005
Title	Delete Note via API and Validate Removal from Database
Objective	Confirm that the note deletion API successfully removes the note record and returns confirmation in the response.
Preconditions	- Application server is running. - Database seeded. - User ID 1 authenticated with Sanctum. - At least one note exists for deletion.

Steps	Step 1: Create a note in the database. Step 2: Send DELETE request to /api/v1/notes/{note->id} with authentication headers. Step 3: Assert API response status is 200 OK and JSON contains {"success": true}. Step 4: Assert that the note record is deleted from the database.
Test Data	- Note created via Note::factory()->create(). - HTTP DELETE request to /api/v1/notes/{id}. Expected values: - HTTP status 200. - JSON response includes {"success": true}. - Note no longer exists in database.
Expected Result	The API responds with HTTP 200 OK and JSON {"success": true}, and the note is removed from the database.
Actual Result	The API responds with HTTP 200 OK and JSON {"success": true}, and the note is removed from the database.
Status	Pass
Severity	High

File: Admin\PaymentMethodTest.txt

Test ID	TC-Adm-PMT-001
Title	Retrieve paginated list of payment methods via API
Objective	Validate the integration between the payment methods API endpoint and the database to ensure that payment methods can be listed successfully.
Preconditions	Application is running. Database is seeded with payment methods and demo data. Authenticated user session is established (Sanctum).
Steps	Step 1: Send a GET request to ‘api/v1/payment-methods?page=1’. Step 2: Assert the response status is HTTP 200 OK.
Test Data	Input: GET request to ‘api/v1/payment-methods?page=1’ Expected: HTTP 200 OK response containing a paginated list of payment methods.
Expected Result	API responds with HTTP 200 OK and returns a paginated list of existing payment methods.
Actual Result	API responds with HTTP 200 OK and returns a paginated list of existing payment methods.
Status	Pass
Severity	High

Test ID	TC-Adm-PMT-002
Title	Create a new payment method via API and persist to database
Objective	Validate the ability of the API to accept new payment method details, create a record, and reflect changes in the database.
Preconditions	Application is running. Database is seeded and connected. Authenticated user session is established.
Steps	Step 1: Prepare payload with name ‘demo name’ and valid company_id. Step 2: Send POST request to ‘api/v1/payment-methods’ with payload. Step 3: Assert response status is HTTP 201 Created. Step 4: Verify that payment_methods table contains the new entry.
Test Data	Input: POST request to ‘api/v1/payment-methods’ with: - name: ‘demo name’ - company_id: [ID of user’s first company] Expected: HTTP 201 Created response and payment_methods table contains new record with specified name and company_id.
Expected Result	API responds with HTTP 201 Created. Database contains a new payment method record with the specified name and company_id.
Actual Result	API responds with HTTP 201 Created. Database contains a new payment method record with the specified name and company_id.
Status	Pass
Severity	High

Test ID	TC-Adm-PMT-003
Title	Store (create) payment method validates inputs using form request
Objective	Verify that the API enforces proper validation by ensuring the PaymentMethodsController ‘store’ action uses the PaymentMethodRequest form request.
Preconditions	Application is running. Required classes and form requests are available.
Steps	Step 1: Assert that the PaymentMethodsController@store action uses PaymentMethodRequest for validation.

Test Data	Input: Validation logic for the ‘store’ action. Expected: The ‘store’ action of PaymentMethodsController uses PaymentMethodRequest for input validation.
Expected Result	PaymentMethodsController@store action uses PaymentMethodRequest, guaranteeing input validation for payment method creation.
Actual Result	PaymentMethodsController@store action uses PaymentMethodRequest, guaranteeing input validation for payment method creation.
Status	Pass
Severity	Medium

Test ID	TC-Adm-PMT-004
Title	Retrieve specific payment method details via API
Objective	Ensure accurate retrieval of a single payment method’s details via API and verify data integrity in the database.
Preconditions	Application is running. Database is seeded. A payment method exists in the database. Authenticated user session.
Steps	Step 1: Create and persist a payment method in the database. Step 2: Send GET request to ‘api/v1/payment-methods/{method_id}’. Step 3: Assert response status is HTTP 200 OK. Step 4: Confirm database contains the payment method with expected id, name, and company_id.
Test Data	Input: GET request to ‘api/v1/payment-methods/{method_id}’ where {method_id} is the id of a newly created payment method. Expected: HTTP 200 OK response containing the payment method’s details matching database record.
Expected Result	API responds with HTTP 200 OK and returns details of the specified payment method. Database record matches the returned data.
Actual Result	API responds with HTTP 200 OK and returns details of the specified payment method. Database record matches the returned data.
Status	Pass
Severity	High

Test ID	TC-Adm-PMT-005
Title	Update existing payment method's details via API and persist changes
Objective	Confirm that API properly updates an existing payment method’s details and reflects changes in the database.
Preconditions	Application is running. Database contains at least one payment method. Authenticated user session.
Steps	Step 1: Create and persist a payment method. Step 2: Send PUT request to ‘api/v1/payment-methods/{method_id}’ with updated name. Step 3: Assert response status is HTTP 200 OK. Step 4: Verify that payment_methods table reflects the updated name.
Test Data	Input: PUT request to ‘api/v1/payment-methods/{method_id}’ with: - name: ‘updated name’ Expected: HTTP 200 OK response. Database reflects updated name for payment method with given id.
Expected Result	API responds with HTTP 200 OK. Database contains updated name for the specified payment method.
Actual Result	API responds with HTTP 200 OK. Database contains updated name for the specified payment method.

Status	Pass
Severity	High

Test ID	TC-Adm-PMT-006
Title	Update payment method validates inputs using form request
Objective	Verify that editing a payment method via API uses the PaymentMethodRequest form request for input validation.
Preconditions	Application is running. Required controller and validation class available.
Steps	Step 1: Assert that PaymentMethodsController@update action uses PaymentMethodRequest for validation.
Test Data	Input: Validation logic for the ‘update’ action. Expected: The ‘update’ action in PaymentMethodsController uses PaymentMethodRequest for input validation.
Expected Result	PaymentMethodsController@update action uses PaymentMethodRequest, ensuring request payload validation when updating payment methods.
Actual Result	PaymentMethodsController@update action uses PaymentMethodRequest, ensuring request payload validation when updating payment methods.
Status	Pass
Severity	Medium

Test ID	TC-Adm-PMT-007
Title	Delete existing payment method via API and remove from database
Objective	Verify the integration between the payment method delete API endpoint and database to ensure that payment methods can be deleted and are fully removed.
Preconditions	Application is running. Database contains at least one payment method. Authenticated user session.
Steps	Step 1: Create and persist a payment method in the database. Step 2: Send DELETE request to ‘api/v1/payment-methods/{method_id}’. Step 3: Assert response status is HTTP 200 OK. Step 4: Verify the payment method record is deleted from the database.
Test Data	Input: DELETE request to ‘api/v1/payment-methods/{method_id}’ where method_id exists. Expected: HTTP 200 OK response and payment method removed from database.
Expected Result	API responds with HTTP 200 OK. The specified payment method record is deleted from the database.
Actual Result	API responds with HTTP 200 OK. The specified payment method record is deleted from the database.
Status	Pass
Severity	High

File: Admin\PaymentTest.txt

Test ID	TC-Adm-PT-001
Title	Retrieve list of payments via API
Objective	Validate that the payments API correctly returns a paginated list of payments, integrating between the API layer and payment data store.
Preconditions	Application running with web server and API enabled Database seeded with demo and core data Authenticated user context via Sanctum
Steps	Step 1: Authenticate as test user and set company headers Step 2: Send GET request to /api/v1/payments?page=1 Step 3: Verify HTTP response status is 200 (OK)
Test Data	GET request to api/v1/payments?page=1 No request payload, relies on seeded payments
Expected Result	API responds with HTTP 200 OK and a paginated JSON list of payments matching current company context.
Actual Result	API responds with HTTP 200 OK and a paginated JSON list of payments matching current company context.
Status	Pass
Severity	High

Test ID	TC-Adm-PT-002
Title	Retrieve specific payment details via API
Objective	Confirm that a payment's details can be fetched by ID, integrating API with payment record retrieval.
Preconditions	Application running and API accessible Database seeded with payments Authenticated user present
Steps	Step 1: Create a new payment in database Step 2: Send GET request to /api/v1/payments/{payment_id} Step 3: Check HTTP status is 200
Test Data	GET request to api/v1/payments/{payment_id} (with a test payment created during test)
Expected Result	API responds with HTTP 200 and returns the details of the specified payment ID as a JSON object.
Actual Result	API responds with HTTP 200 and returns the details of the specified payment ID as a JSON object.
Status	Pass
Severity	High

Test ID	TC-Adm-PT-003
Title	Create a payment and verify persistence
Objective	Ensure payments can be created via the API and the new entry is persisted in the database.
Preconditions	Application running and API accessible Database seeded Test invoice exists for payment association Authenticated user present
Steps	Step 1: Create invoice record Step 2: Prepare payment payload with invoice info Step 3: POST payment data to /api/v1/payments Step 4: Verify response status is OK (200) Step 5: Assert new payment exists in database with correct values

Test Data	POST request to api/v1/payments with payload: payment_number: "PAY-000001" amount: 100 exchange_rate: 1 invoice_id: [test invoice id] Other payment fields populated from factory Expected payment record persists in DB
Expected Result	API returns HTTP 200 OK; payment record is found in database with matching payment_number, customer_id, amount, company_id.
Actual Result	API returns HTTP 200 OK; payment record is found in database with matching payment_number, customer_id, amount, company_id.
Status	Pass
Severity	High

Test ID	TC-Adm-PT-004
Title	Validate payment creation enforces form request
Objective	Verify the integration between the controller and form request, ensuring payment creation uses PaymentRequest validation.
Preconditions	Application running Payment creation endpoint available Controller and PaymentRequest classes registered
Steps	Step 1: Execute assertion that PaymentsController@store uses PaymentRequest for validation
Test Data	N/A (behavioral test verifying class-level integration, no request payload)
Expected Result	Assertion passes; PaymentsController@store method integrates with PaymentRequest for validation.
Actual Result	Assertion passes; PaymentsController@store method integrates with PaymentRequest for validation.
Status	Pass
Severity	Medium

Test ID	TC-Adm-PT-005
Title	Update an existing payment and verify changes
Objective	Confirm that updating a payment via API persists updated fields in the database, integrating the full update flow.
Preconditions	Application running Database seeded Test invoice and payment exist Authenticated user context
Steps	Step 1: Create invoice and associated payment Step 2: Prepare updated payment data Step 3: Send PUT request to /api/v1/payments/{payment_id} with new data Step 4: Assert response is OK (200) Step 5: Assert database has payment with updated fields
Test Data	PUT request to api/v1/payments/{payment_id} with new payment data (from factory) Fields: payment_number, customer_id, amount, exchange_rate, invoice_id
Expected Result	API returns HTTP 200 OK; the payment record in the database is updated with new payment_number, customer_id, and amount.
Actual Result	API returns HTTP 200 OK; the payment record in the database is updated with new payment_number, customer_id, and amount.
Status	Pass
Severity	High

Test ID	TC-Adm-PT-006
Title	Validate payment update enforces form request
Objective	Ensure the PaymentsController@update method uses PaymentRequest for validation, confirming controller/request integration.
Preconditions	Application running Payment update endpoint available Controller and PaymentRequest classes present
Steps	Step 1: Assert PaymentsController@update uses PaymentRequest for request validation
Test Data	N/A (assert integration; no direct API call or payload)
Expected Result	Assertion passes; PaymentsController@update integrates with PaymentRequest for validation.
Actual Result	Assertion passes; PaymentsController@update integrates with PaymentRequest for validation.
Status	Pass
Severity	Medium

Test ID	TC-Adm-PT-007
Title	Search payments using filters via API
Objective	Validate that payment search functionality works with filter parameters, correctly integrating API with payment search.
Preconditions	Application running and API accessible Database seeded with payments containing specific payment numbers and customer info Authenticated user present
Steps	Step 1: Construct query string with filters Step 2: Send GET request to /api/v1/payments?{filters} Step 3: Assert HTTP status OK (200)
Test Data	GET request to api/v1/payments with query string filters: page=1 limit=15 search=doe payment_number=PAY-000001 payment_mode=OTHER
Expected Result	API returns HTTP 200 OK and responds with filtered payments list matching the search and filters.
Actual Result	API returns HTTP 200 OK and responds with filtered payments list matching the search and filters.
Status	Pass
Severity	Medium

Test ID	TC-Adm-PT-008
Title	Send payment notification email to customer
Objective	Ensure payment notification emails are sent when requested via API, integrating payment data with mail services.
Preconditions	Application running and mail subsystem configured or faked Payment record exists Authenticated user

Steps	Step 1: Fake mail delivery Step 2: Create payment record Step 3: Send POST request to send payment email endpoint Step 4: Assert JSON response contains 'success' => true Step 5: Assert SendPaymentMail was sent
Test Data	POST request to api/v1/payments/{payment_id}/send with payload: subject: 'test' body: 'test' from: 'john@example.com' to: 'doe@example.com'
Expected Result	API returns success (success: true); payment mail notification sent to specified recipient.
Actual Result	API returns success (success: true); payment mail notification sent to specified recipient.
Status	Pass
Severity	High

Test ID	TC-Adm-PT-009
Title	Bulk delete payments via API and verify removal
Objective	Validate that multiple payments can be deleted in a single request and records are removed from the database.
Preconditions	Application running and API accessible At least five payment records exist Authenticated user context
Steps	Step 1: Create 5 payment records Step 2: Send POST request to /api/v1/payments/delete with payment IDs Step 3: Assert JSON response contains 'success' => true
Test Data	POST request to api/v1/payments/delete with payload: ids: [array of payment IDs to delete]
Expected Result	API returns JSON indicating success (success: true); specified payments are deleted from database.
Actual Result	API returns JSON indicating success (success: true); specified payments are deleted from database.
Status	Pass
Severity	High

Test ID	TC-Adm-PT-010
Title	Create payment without invoice and verify persistence
Objective	Validate that a payment can be created without being linked to an invoice, and is correctly stored.
Preconditions	Application running Database seeded No invoice record required for payment creation Authenticated user present
Steps	Step 1: Prepare raw payment payload (no invoice_id) Step 2: Send POST request to /api/v1/payments Step 3: Assert HTTP OK (200) Step 4: Assert payment record exists in database
Test Data	POST request to api/v1/payments with payload: payment_number: "PAY-000001" exchange_rate: 1 Other payment fields from factory Expected persistence in database
Expected Result	API returns HTTP 200 OK; payment record stored with correct payment_number, customer_id, amount, company_id.

Actual Result	API returns HTTP 200 OK; payment record stored with correct payment_number, customer_id, amount, company_id.
Status	Pass
Severity	Medium

Test ID	TC-Adm-PT-011
Title	Create payment with invoice association and verify linkage
Objective	Confirm that payments created with an invoice are stored with a correct invoice association.
Preconditions	Application running Database seeded Invoice record available Authenticated user context
Steps	Step 1: Create invoice Step 2: Prepare payment payload with invoice linkage Step 3: Send POST to /api/v1/payments Step 4: Assert HTTP OK (200) Step 5: Assert database matches payment info, including invoice_id
Test Data	POST request to api/v1/payments with payload: payment_number: "PAY-000001" invoice_id: [test invoice id] amount: [invoice due amount] exchange_rate: 1 Other fields from factory
Expected Result	API returns HTTP 200 OK; payment record is stored with correct payment_number, customer_id, invoice_id, amount, company_id.
Actual Result	API returns HTTP 200 OK; payment record is stored with correct payment_number, customer_id, invoice_id, amount, company_id.
Status	Pass
Severity	High

Test ID	TC-Adm-PT-012
Title	Create payment for partially paid invoice and validate invoice/payment states
Objective	Validate that a payment can be created for an invoice in a partially paid state, and invoice/payment data are updated.
Preconditions	Application running Database seeded Invoice exists with partial payment context (sub_total, base fields, discounts, etc.) Authenticated user
Steps	Step 1: Create invoice with partial payment fields Step 2: Prepare payment payload linked to invoice Step 3: Send POST to /api/v1/payments Step 4: Assert HTTP OK (200) Step 5: Assert payment found in DB (matching payment_number, customer_id, amount) Step 6: Assert invoice updated in DB with correct fields (invoice_number, total, customer_id, exchange_rate, base_total, paid_status)
Test Data	POST request to api/v1/payments with payload: invoice_id: [test invoice id] customer_id: [from invoice] exchange_rate: [from invoice] amount: 100 currency_id: [invoice currency] With various invoice fields set for partial payment scenario

Expected Result	API returns HTTP 200 OK; payment and invoice records both reflect correct values, including updated invoice paid_status and payment fields.
Actual Result	API returns HTTP 200 OK; payment and invoice records both reflect correct values, including updated invoice paid_status and payment fields.
Status	Pass
Severity	High

File: Admin\RecurringInvoiceTest.txt

Test ID	TC-Adm-RIT-001
Title	Retrieve List of Recurring Invoices via API
Objective	Validate that the API endpoint returns a paginated list of recurring invoices stored in the database.
Preconditions	Application server is running with API accessible. Database properly seeded with at least one recurring invoice record. User is authenticated and has a valid company context for requests.
Steps	Step 1: Seed the database and create one recurring invoice record. Step 2: Send a GET request to the recurring invoices listing endpoint with pagination. Step 3: Verify the HTTP status in the response is 200 OK.
Test Data	GET request to: api/v1/recurring-invoices?page=1 Expected HTTP status: 200 OK
Expected Result	API responds with HTTP 200 and a paginated list containing at least one recurring invoice from the database.
Actual Result	API responds with HTTP 200 and a paginated list containing at least one recurring invoice from the database.
Status	Pass
Severity	High

Test ID	TC-Adm-RIT-002
Title	Validate Recurring Invoice Creation Request Uses Form Request Validation
Objective	Ensure the recurring invoice creation API uses proper form request validation (business and data rules) for POST submissions.
Preconditions	Application server is running. RecurringInvoiceController and RecurringInvoiceRequest classes are loaded. Test environment supports request validation.
Steps	Step 1: Assert that the RecurringInvoiceController@store action uses RecurringInvoiceRequest for validation. Step 2: Verify internal controller configuration for form request usage.
Test Data	Controller: RecurringInvoiceController Action: store Form request: RecurringInvoiceRequest
Expected Result	The store method of the controller is configured to use RecurringInvoiceRequest, ensuring business and validation logic is applied during invoice creation.
Actual Result	The store method of the controller is configured to use RecurringInvoiceRequest, ensuring business and validation logic is applied during invoice creation.
Status	Pass
Severity	High

Test ID	TC-Adm-RIT-003
Title	Create Recurring Invoice via API and Persist to Database
Objective	Verify that the recurring invoice creation endpoint saves new records and associated invoice items to the database.
Preconditions	Application server running, API accessible. Database seeded. User authenticated. Invoice items and recurring invoice models available.

Steps	Step 1: Create a raw recurring invoice payload including one invoice item. Step 2: Send a POST request to the recurring invoice creation endpoint. Step 3: Assert that the API returns HTTP 201 Created. Step 4: Check the database contains the newly created recurring invoice with expected values (e.g., frequency).
Test Data	POST to: api/v1/recurring-invoices Payload: Recurring invoice fields with one invoice item in 'items'. Expected response status: 201 Created Expected DB values: recurring_invoices record with the given frequency
Expected Result	API returns HTTP 201 Created. New recurring invoice is present in the database with correct frequency and associated items.
Actual Result	API returns HTTP 201 Created. New recurring invoice is present in the database with correct frequency and associated items.
Status	Pass
Severity	High

Test ID	TC-Adm-RIT-004
Title	Retrieve Specific Recurring Invoice by ID via API
Objective	Validate that the API returns details of an individual recurring invoice identified by its ID.
Preconditions	Application server running. Database seeded, recurring invoice exists. User is authenticated.
Steps	Step 1: Create a recurring invoice in the database. Step 2: Send a GET request to retrieve details of this recurring invoice by its ID. Step 3: Assert the API response is HTTP 200 OK.
Test Data	GET to: api/v1/recurring-invoices/{recurringInvoiceId} Expected HTTP status: 200 OK
Expected Result	API responds with HTTP 200 OK and the details of the requested recurring invoice.
Actual Result	API responds with HTTP 200 OK and the details of the requested recurring invoice.
Status	Pass
Severity	High

Test ID	TC-Adm-RIT-005
Title	Validate Update Request for Recurring Invoice Uses Form Request Validation
Objective	Ensure that updates for recurring invoices use the designated form request for validation, enforcing business/data consistency.
Preconditions	Application server running. RecurringInvoiceController and RecurringInvoiceRequest classes available.
Steps	Step 1: Assert that the RecurringInvoiceController@update action uses RecurringInvoiceRequest for validation. Step 2: Validate controller configuration.
Test Data	Controller: RecurringInvoiceController Action: update Form Request: RecurringInvoiceRequest
Expected Result	The update method of the controller is configured to use RecurringInvoiceRequest for request validation.
Actual Result	The update method of the controller is configured to use RecurringInvoiceRequest for request validation.
Status	Pass

Severity	High

Test ID	TC-Adm-RIT-006
Title	Update Existing Recurring Invoice via API and Persist Changes
Objective	Confirm that updating an existing recurring invoice via the API changes the stored record in the database.
Preconditions	Application server running. Database seeded with recurring invoice. User authenticated. Invoice items and recurring invoice models available.
Steps	Step 1: Create a recurring invoice in the database. Step 2: Prepare new data payload (recurring invoice and one invoice item). Step 3: Send a PUT request to update the recurring invoice. Step 4: Assert the API returns HTTP 200 OK. Step 5: Validate the database contains the updated recurring invoice with new values.
Test Data	PUT to: api/v1/recurring-invoices/{recurringInvoiceId} Payload: New recurring invoice fields and invoice item. Expected response status: 200 OK Expected DB values: recurring_invoices record updated with new frequency
Expected Result	API responds with HTTP 200 OK, and the recurring invoice record in the database is updated with provided changes (e.g., frequency).
Actual Result	API responds with HTTP 200 OK, and the recurring invoice record in the database is updated with provided changes (e.g., frequency).
Status	Pass
Severity	High

Test ID	TC-Adm-RIT-007
Title	Delete Multiple Recurring Invoices via API and Verify Database Removal
Objective	Verify that a bulk delete request removes multiple recurring invoices from the database and responds accurately.
Preconditions	Application server running. Database seeded with at least three recurring invoices. User authenticated.
Steps	Step 1: Seed the database with three recurring invoices. Step 2: Send a POST request to the bulk delete endpoint with all invoice IDs. Step 3: Assert the API response is OK and contains 'success' => true. Step 4: Verify each recurring invoice is deleted from the database.
Test Data	POST to: api/v1/recurring-invoices/delete Payload: Array of three recurring invoice IDs Expected response: HTTP 200 OK; JSON 'success' => true
Expected Result	API returns HTTP 200 OK and 'success' => true. All specified recurring invoices are deleted from the database.
Actual Result	API returns HTTP 200 OK and 'success' => true. All specified recurring invoices are deleted from the database.
Status	Pass
Severity	High

Test ID	TC-Adm-RIT-008
Title	Calculate Invoice Frequency Schedule via API Endpoint
Objective	Validate that the API correctly computes the occurrence dates for recurring invoices based on given frequency and start date.

Preconditions	Application server running. API endpoint for frequency calculation accessible.
Steps	Step 1: Prepare frequency and start date values. Step 2: Send GET request to frequency calculation endpoint with above data. Step 3: Assert API returns HTTP 200 OK.
Test Data	GET to: api/v1/recurring-invoice-frequency?frequency=* * 2 * *&starts;_at={current_date} Expected response: HTTP 200 OK
Expected Result	API responds with HTTP 200 OK, calculating and returning occurrence dates or computation results as per provided frequency and start date.
Actual Result	API responds with HTTP 200 OK, calculating and returning occurrence dates or computation results as per provided frequency and start date.
Status	Pass
Severity	Medium

File: Admin\RoleTest.txt

Test ID	TC-Adm-RT-001
Title	Verify User Creation with Super Admin Role via API
Objective	To validate that the integration between the user creation API endpoint and the database works as expected when creating a new user with the "super admin" role. This test ensures the API successfully creates the user and persists the correct data in the database with the required role association.
Preconditions	- Application is running and accessible. - Database is seeded with core and demo data (DatabaseSeeder, DemoSeeder). - User authentication is active via Laravel Sanctum, acting as an existing user (ID=1). - Request headers correctly set to specify target company context.
Steps	Step 1: Seed the database with required data (DatabaseSeeder and DemoSeeder). Step 2: Authenticate as user with ID=1 and set company header to the user's first company. Step 3: Send a POST JSON request to 'api/v1/users' with user details including super admin role assignment. Step 4: Assert that the API returns HTTP 201 status, indicating successful creation. Step 5: Assert that the database contains a user record with the specified email and name.

Test Data	Input:
	{
	"email": "loremipsum@gmail.com",
	"name": "lorem",
	"password": "lorem@123",
	"companies": [
	{
	"role": "super admin",
	"id": 1
	}
	]
	}
	Expected Values:
	- HTTP status: 201 (Created)
	- Database should contain a user record with:
	email: "loremipsum@gmail.com"
	name: "lorem"

Expected Result	User is created via the API with the specified email and name, assigned to company ID=1 with "super admin" role. API responds with HTTP 201 status. Database contains a new user record matching the input data (email and name fields).
Actual Result	User is created via the API with the specified email and name, assigned to company ID=1 with "super admin" role. API responds with HTTP 201 status. Database contains a new user record matching the input data (email and name fields).
Status	Pass
Severity	High

File: Admin\TaxTypeTest.txt

Test ID	TC-Adm-TTT-001
Title	Retrieve list of tax types via API
Objective	Validate that the tax types endpoint returns all available tax types from the database for an authenticated admin user.
Preconditions	- Application is running in a test environment. - Database is seeded with required data using DatabaseSeeder and DemoSeeder. - An authenticated admin user (User::find(1)) is acting via Laravel Sanctum. - Company header is set correctly for multi-company context.
Steps	1. Authenticate as admin user using Sanctum. 2. Send GET request to 'api/v1/tax-types'. 3. Verify that the response status is HTTP 200 OK.
Test Data	Input: GET request to 'api/v1/tax-types' endpoint. Expected: HTTP 200 OK response containing list of tax types.
Expected Result	The response status is HTTP 200 OK and a list of existing tax types is returned, confirming successful retrieval of tax types for the authenticated admin user.
Actual Result	The response status is HTTP 200 OK and a list of existing tax types is returned, confirming successful retrieval of tax types for the authenticated admin user.
Status	Pass
Severity	High

Test ID	TC-Adm-TTT-002
Title	Create new tax type through API
Objective	Verify that the system allows an authenticated admin user to create a new tax type via the API and persists it in the database.
Preconditions	- Application running with all migrations and seeders up-to-date. - Database seeded, company context provided. - Authenticated admin user session established.
Steps	1. Prepare tax type data using model factory. 2. Send POST request with tax type data to 'api/v1/tax-types'. 3. Assert presence of new tax type record in the database.
Test Data	Input: Tax type attributes as per factory returned by TaxType::factory()->raw(). Expected: Tax type record with input attributes created in 'tax_types' table.
Expected Result	New tax type is successfully created in the database, confirming the POST API endpoint works and the record matches the provided attributes.
Actual Result	New tax type is successfully created in the database, confirming the POST API endpoint works and the record matches the provided attributes.
Status	Pass
Severity	High

Test ID	TC-Adm-TTT-003
Title	Validate tax type creation uses correct form request
Objective	Ensure the TaxTypesController::store action integrates correctly with TaxTypeRequest for form validation on tax type creation.
Preconditions	- Application running with valid controller and request classes. - Autoloading and dependency injection is functional.
Steps	1. Inspect TaxTypesController::store action. 2. Assert that form request validation is wired to use TaxTypeRequest.

Test Data	Input: N/A (test inspects controller action configuration). Expected: TaxTypesController::store uses TaxTypeRequest for validation.
Expected Result	TaxTypesController::store is configured to use TaxTypeRequest for validating input data when creating a tax type.
Actual Result	TaxTypesController::store is configured to use TaxTypeRequest for validating input data when creating a tax type.
Status	Pass
Severity	Medium

Test ID	TC-Adm-TTT-004
Title	Retrieve single tax type details via API
Objective	Verify that the API returns correct tax type details for a specific tax type ID provided by an authenticated admin user.
Preconditions	- Application and database seeded. - Company context and authenticated admin user setup. - At least one tax type exists in the database.
Steps	1. Create a tax type entry using model factory. 2. Send GET request to 'api/v1/tax-types/{id}' using the created tax type's ID. 3. Assert response status is HTTP 200 OK.
Test Data	Input: GET request to 'api/v1/tax-types/{id}', where {id} is an existing tax type's ID. Expected: HTTP 200 OK and details of the specified tax type.
Expected Result	HTTP 200 OK is returned and the response body contains details of the requested tax type, matching the record in the database.
Actual Result	HTTP 200 OK is returned and the response body contains details of the requested tax type, matching the record in the database.
Status	Pass
Severity	High

Test ID	TC-Adm-TTT-005
Title	Update tax type information via API
Objective	Confirm that the system allows admin users to update an existing tax type using the API and persists the changes correctly.
Preconditions	- Application is running and seeded. - User authentication and company context provided. - At least one tax type exists in the database.
Steps	1. Create initial tax type record. 2. Generate updated tax type attributes with model factory. 3. Send PUT request to endpoint with new attributes. 4. Assert that the response status is HTTP 200 OK.
Test Data	Input: PUT request to 'api/v1/tax-types/{id}' with updated attributes from TaxType::factory()->raw(). Expected: HTTP 200 OK and tax type updates applied in the database.
Expected Result	Tax type is updated with new attributes, HTTP 200 OK is returned, and the database reflects the changes.
Actual Result	Tax type is updated with new attributes, HTTP 200 OK is returned, and the database reflects the changes.
Status	Pass
Severity	High

Test ID	TC-Adm-TTT-006
Title	Validate tax type update uses form request

Objective	Ensure that the update action in TaxTypesController integrates with TaxTypeRequest for input validation during tax type updates.
Preconditions	- Application is running. - TaxTypesController and TaxTypeRequest classes are properly implemented.
Steps	1. Inspect TaxTypesController::update action. 2. Assert that form request validation is set to use TaxTypeRequest.
Test Data	Input: N/A (controller configuration check). Expected: TaxTypesController::update uses TaxTypeRequest for validation.
Expected Result	TaxTypesController::update action is configured to use TaxTypeRequest for validating inputs during update operations.
Actual Result	TaxTypesController::update action is configured to use TaxTypeRequest for validating inputs during update operations.
Status	Pass
Severity	Medium

Test ID	TC-Adm-TTT-007
Title	Delete tax type via API
Objective	Confirm that an authenticated admin can delete a tax type using the API, and the record is permanently removed from the database.
Preconditions	- Application running and fully seeded. - Authenticated admin user scoped to correct company. - At least one tax type is present in the database.
Steps	1. Create a tax type record. 2. Send DELETE request to 'api/v1/tax-types/{id}'. 3. Assert HTTP 200 OK status and JSON body contains 'success' => true. 4. Confirm the tax type record is deleted from the database.
Test Data	Input: DELETE request to 'api/v1/tax-types/{id}' for an existing tax type. Expected: HTTP 200 OK response, success: true in JSON body, and record removed from 'tax_types' table.
Expected Result	API responds with HTTP 200 OK and 'success' => true; the tax type record is deleted and absent from the database.
Actual Result	API responds with HTTP 200 OK and 'success' => true; the tax type record is deleted and absent from the database.
Status	Pass
Severity	High

Test ID	TC-Adm-TTT-008
Title	Create tax type with negative percent value
Objective	Verify that the system permits creating tax types with negative percent values and correctly persists them via API.
Preconditions	- Application running and seeded. - Valid admin user session. - Company context header present.
Steps	1. Generate tax type data with 'percent' set to -9.99. 2. Send POST request to create a new tax type with this data. 3. Assert response status is HTTP 201 Created. 4. Verify the new tax type with negative percent exists in database.
Test Data	Input: POST request to 'api/v1/tax-types' with tax type data including 'percent' => -9.99. Expected: HTTP 201 Created, and tax type with -9.99 percent present in 'tax_types' table.
Expected Result	New tax type with a percent value of -9.99 is created, HTTP 201 returned, and record exists in database.

Actual Result	New tax type with a percent value of -9.99 is created, HTTP 201 returned, and record exists in database.
Status	Pass
Severity	Medium

File: Admin\UnitTest.txt

Test ID	TC-Adm-UT-001
Title	Retrieve Units List via API
Objective	Validate that the Units API endpoint returns a paginated list of units and is accessible via authenticated requests.
Preconditions	- Application running with necessary services. - Database seeded with standard and demo data. - Authenticated user present with valid company context.
Steps	Step 1: Send GET request to api/v1/units?page=1 as authenticated user. Step 2: Verify that the response status is 200 OK.
Test Data	Input: GET request to api/v1/units?page=1 Expected: HTTP 200 OK response; units list present in response.
Expected Result	The API returns a 200 OK status and a JSON list of available units for the authenticated user's company.
Actual Result	The API returns a 200 OK status and a JSON list of available units for the authenticated user's company.
Status	Pass
Severity	Medium

Test ID	TC-Adm-UT-002
Title	Create New Unit via API
Objective	Validate that the Units API endpoint allows creation of a new Unit and persists the correct data in the database.
Preconditions	- Application running with necessary services. - Database seeded with standard and demo data. - Authenticated user present with valid company context.
Steps	Step 1: Send POST request with unit data to api/v1/units. Step 2: Verify the response status is 201 Created. Step 3: Assert the units table contains a unit matching the submitted data.
Test Data	Input: POST request to api/v1/units, payload: { "name": "unit name", "company_id": [company_id] } Expected: HTTP 201 Created; unit with the given data exists in the database.
Expected Result	API returns a 201 Created status and the database contains the new unit with the specified name and company_id.
Actual Result	API returns a 201 Created status and the database contains the new unit with the specified name and company_id.
Status	Pass
Severity	High

Test ID	TC-Adm-UT-003
Title	Unit Creation Validates with Form Request
Objective	Confirm that the UnitsController's store action uses UnitRequest form validation as part of the request handling process.
Preconditions	- Application running with necessary services. - Relevant controller and request classes present and autoloaded.
Steps	Step 1: Assert that UnitsController@store is configured to use UnitRequest form validation.
Test Data	Input: Simulated call to UnitsController@store Expected: Store action invokes UnitRequest form validation.

Expected Result	The UnitsController store action uses UnitRequest for form validation on incoming requests.
Actual Result	The UnitsController store action uses UnitRequest for form validation on incoming requests.
Status	Pass
Severity	Medium

Test ID	TC-Adm-UT-004
Title	Retrieve Unit Details via API
Objective	Validate that the Units API endpoint returns details for an individual unit based on its ID, confirming database integrity.
Preconditions	- Application running. - Database seeded. - Authenticated user context. - At least one Unit exists in the database.
Steps	Step 1: Create a Unit via factory. Step 2: Send GET request to api/v1/units/{unit_id}. Step 3: Assert response status is 200 OK. Step 4: Confirm the units table contains record with given id and name.
Test Data	Input: GET request to api/v1/units/{unit_id} Expected: HTTP 200 OK; response contains unit data matching {id, name}.
Expected Result	API returns a 200 OK status with the unit details; database contains the unit with the correct id and name.
Actual Result	API returns a 200 OK status with the unit details; database contains the unit with the correct id and name.
Status	Pass
Severity	Medium

Test ID	TC-Adm-UT-005
Title	Update Unit via API
Objective	Validate that the Units API endpoint allows updating a Unit's name and that the changes propagate correctly to the database.
Preconditions	- Application running. - Database seeded. - Authenticated user context. - At least one Unit exists in the database.
Steps	Step 1: Create a Unit via factory. Step 2: Send PUT request with new name to api/v1/units/{unit_id}. Step 3: Assert response status is 200 OK. Step 4: Verify database contains unit with updated name and original id.
Test Data	Input: PUT request to api/v1/units/{unit_id}, payload: { "name": "new name" } Expected: HTTP 200 OK; unit record in database has updated name.
Expected Result	API returns a 200 OK status, and the database record for the unit reflects the updated name.
Actual Result	API returns a 200 OK status, and the database record for the unit reflects the updated name.
Status	Pass
Severity	High

Test ID	TC-Adm-UT-006
Title	Unit Update Validates with Form Request

Objective	Ensure that the UnitsController's update action leverages UnitRequest for request validation when updating units.
Preconditions	- Application running. - Relevant controller and request classes present and autoloaded.
Steps	Step 1: Assert that UnitsController@update is configured to use UnitRequest for form validation.
Test Data	Input: Simulated call to UnitsController@update Expected: Update action invokes UnitRequest form validation.
Expected Result	The UnitsController update action uses UnitRequest to validate incoming update requests.
Actual Result	The UnitsController update action uses UnitRequest to validate incoming update requests.
Status	Pass
Severity	Medium

Test ID	TC-Adm-UT-007
Title	Delete Unit via API
Objective	Validate that the Units API endpoint allows deletion of an existing unit and removes it from the database.
Preconditions	- Application running. - Database seeded. - Authenticated user context. - At least one Unit exists in the database.
Steps	Step 1: Create a Unit via factory. Step 2: Send DELETE request to api/v1/units/{unit_id}. Step 3: Assert response status is 200 OK. Step 4: Assert the unit record is deleted from the database.
Test Data	Input: DELETE request to api/v1/units/{unit_id} Expected: HTTP 200 OK; unit record is deleted from database.
Expected Result	API returns a 200 OK status, and the unit is no longer present in the database.
Actual Result	API returns a 200 OK status, and the unit is no longer present in the database.
Status	Pass
Severity	High

File: Admin\UserTest.txt

Test ID	TC-Adm-UT-001
Title	Retrieve All Users via API Endpoint
Objective	Validate that the Users API endpoint correctly returns a list of all users and is accessible for authorized super admin users.
Preconditions	Application is running. Database is seeded with initial data (DatabaseSeeder and DemoSeeder). Super admin user exists and is authenticated via Sanctum. API endpoint /api/v1/users is accessible and configured.
Steps	Step 1: Authenticate as super admin user. Step 2: Send a GET request to /api/v1/users. Step 3: Verify that response status is 200 OK.
Test Data	Authenticated super admin user. Expected HTTP response status: 200 OK.
Expected Result	The API responds with HTTP 200 OK, indicating successful retrieval of the users list.
Actual Result	The API responds with HTTP 200 OK, indicating successful retrieval of the users list.
Status	Pass
Severity	High

Test ID	TC-Adm-UT-002
Title	Validate Store User Endpoint Uses Form Request for Input Validation
Objective	Ensure that the UsersController@store action uses the UserRequest form request class for appropriate input validation during user creation.
Preconditions	Application is running. Database is seeded (DatabaseSeeder, DemoSeeder). UserController and UserRequest exist in the codebase.
Steps	Step 1: Confirm application environment is set up and seeded. Step 2: Verify that UsersController@store action utilizes UserRequest for validation.
Test Data	Targeted controller: UsersController Method: store Form Request: UserRequest
Expected Result	UsersController@store action is found to use UserRequest, ensuring all input data is validated before storing new users.
Actual Result	UsersController@store action is found to use UserRequest, ensuring all input data is validated before storing new users.
Status	Pass
Severity	High

Test ID	TC-Adm-UT-003
Title	Retrieve Specific User Details via API
Objective	Validate the integration between the Users API and the database by ensuring the endpoint correctly returns details for a given user.
Preconditions	Application is running. Database is seeded (DatabaseSeeder, DemoSeeder). A user exists in the database (created via factory). Authenticated as super admin user via Sanctum.
Steps	Step 1: Create a new user using User::factory()->create(). Step 2: Send GET request to /api/v1/users/{user_id}. Step 3: Verify response status is 200 OK.

Test Data	User created with a unique ID. Expected HTTP response status: 200 OK.
Expected Result	The API responds with HTTP 200 OK for /api/v1/users/{user_id}, delivering details of the requested user.
Actual Result	The API responds with HTTP 200 OK for /api/v1/users/{user_id}, delivering details of the requested user.
Status	Pass
Severity	High

Test ID	TC-Adm-UT-004
Title	Validate Update User Endpoint Uses Form Request for Input Validation
Objective	Ensure that the UsersController@update action uses the UserRequest form request class for appropriate input validation during user update operations.
Preconditions	Application is running. Database is seeded (DatabaseSeeder, DemoSeeder). UserController and UserRequest exist in the codebase.
Steps	Step 1: Confirm application environment is set up and seeded. Step 2: Verify that UsersController@update action utilizes UserRequest for validation.
Test Data	Targeted controller: UsersController Method: update Form Request: UserRequest
Expected Result	UsersController@update action is confirmed to use UserRequest, enforcing proper validation of input data before updating user information.
Actual Result	UsersController@update action is confirmed to use UserRequest, enforcing proper validation of input data before updating user information.
Status	Pass
Severity	High

File: CompanySettingTest.txt

Test ID	TC-CST-001
Title	Verify CompanySetting Association to Company
Objective	Validate that a CompanySetting record is properly linked to an existing Company, ensuring relational integrity between settings and company records.
Preconditions	Application is running in test environment. Database is seeded with fundamental and demo data via DatabaseSeeder and DemoSeeder. Company and CompanySetting models are available and functioning.
Steps	Step 1: Create a CompanySetting instance using the factory. Step 2: Assert that the Company relationship of the CompanySetting exists in the database.
Test Data	Input: A CompanySetting record created via factory. Expected: The related Company record exists and is properly associated with the CompanySetting.
Expected Result	The CompanySetting has a valid associated Company, confirmed by an active relationship in the database.
Actual Result	The CompanySetting has a valid associated Company, confirmed by an active relationship in the database.
Status	Pass
Severity	High

Test ID	TC-CST-002
Title	Set and Retrieve Single CompanySetting Value
Objective	Ensure that a new CompanySetting value can be set for a specific Company, and that the value can be retrieved accurately, verifying correct integration between the setting module and company records.
Preconditions	Application is running in test environment. Database is seeded with fundamental and demo data via DatabaseSeeder and DemoSeeder. Company and CompanySetting models are available and functioning.
Steps	Step 1: Generate a random key and value. Step 2: Create a new Company instance. Step 3: Set the CompanySetting for the given key and Company ID. Step 4: Retrieve the CompanySetting for the same key and Company ID. Step 5: Assert that the retrieved value matches the one originally set.
Test Data	Input: Randomly generated key (faker()->name) and value (faker()->word); newly created Company instance. Expected: The value set for the given key and Company is retrievable and matches the original value.
Expected Result	The retrieved CompanySetting value matches the value that was set for the given key and company.
Actual Result	The retrieved CompanySetting value matches the value that was set for the given key and company.
Status	Pass
Severity	High

Test ID	TC-CST-003
Title	Batch Retrieve Company Setting Values by Key Array
Objective	Validate that multiple CompanySetting keys can be fetched in a batch for a specific Company, verifying that the values returned match those set and ensuring correct integration of batch retrieval.

Preconditions	Application is running in test environment. Database is seeded with fundamental and demo data via DatabaseSeeder and DemoSeeder. Company and CompanySetting models are available and functioning.
Steps	Step 1: Generate a random key and value. Step 2: Create a new Company instance. Step 3: Set the CompanySetting for the given key and Company ID. Step 4: Retrieve CompanySetting values using getSettings with an array containing the key. Step 5: Assert that the returned array matches the original key-value pair.
Test Data	Input: Randomly generated key (faker()->name), value (faker()->word), and newly created Company instance. Expected: The returned array contains the key-value pair as set previously.
Expected Result	The returned collection contains the key-value pair that was originally set for the company.
Actual Result	The returned collection contains the key-value pair that was originally set for the company.
Status	Pass
Severity	Medium

File: CompanyTest.txt

Test ID	TC-CT-001
Title	Verify Company-Customers Relationship
Objective	Ensure that when a company is created, it can correctly be associated with multiple customers and the database reflects this relationship.
Preconditions	- Application is running with necessary dependencies. - Database is seeded using DatabaseSeeder and DemoSeeder. - Laravel Eloquent models are available.
Steps	Step 1: Create a company instance using the factory with the hasCustomers() method. Step 2: Verify using Eloquent that the company's customers() relationship exists (detects at least one customer).
Test Data	- Input: New Company instance created with associated customers via factory (hasCustomers). - Expected Value: The company has at least one associated customer; the 'customers' relation exists in the database.
Expected Result	The company is successfully created and has at least one associated customer; the customers() relationship for the company returns existing records and integration between models and the database is validated.
Actual Result	The company is successfully created and has at least one associated customer; the customers() relationship for the company returns existing records and integration between models and the database is validated.
Status	Pass
Severity	High

Test ID	TC-CT-002
Title	Verify Company-Settings Relationship and Count Integrity
Objective	Confirm that a company can have multiple settings, validate the total count of settings, and ensure the integrity of the company-settings relationship within the database.
Preconditions	- Application is running with required services. - Database seeded with DatabaseSeeder and DemoSeeder. - Laravel Eloquent models loaded.
Steps	Step 1: Create a company instance using the factory with hasSettings(5) method (associate 5 settings). Step 2: Assert that the company's settings collection has exactly five items. Step 3: Assert that the settings() relationship exists and is not empty.
Test Data	- Input: Company instance created via factory with five associated settings (hasSettings(5)). - Expected Value: The company has exactly five settings; settings relationship exists and is populated.
Expected Result	A company can be created and linked with exactly five settings; the settings relationship is present and populated with five records, confirming correct integration and integrity of related data.
Actual Result	A company can be created and linked with exactly five settings; the settings relationship is present and populated with five records, confirming correct integration and integrity of related data.
Status	Pass
Severity	Medium

Test ID	TC-CT-003
Title	Verify Company-Users Relationship as a Collection

Objective	Validate that a company is associated with multiple users, and ensure that the collection returned by the company's user relationship is correctly implemented as an Eloquent Collection.
Preconditions	- Application is running and all dependencies are functional. - Database is seeded using DatabaseSeeder and DemoSeeder. - Eloquent models for Company and User are available.
Steps	Step 1: Create a company instance using the factory with hasUsers(5) to create and associate five users. Step 2: Assert that the company's users relationship returns an instance of Illuminate\Database\Eloquent\Collection.
Test Data	- Input: New Company instance created via factory with five associated users (hasUsers(5)). - Expected Value: The company's users property is an instance of Eloquent Collection containing all associated users.
Expected Result	On creating a company with multiple users, the users relationship is returned as an Eloquent Collection and contains all associated users, validating the integrity and proper setup of the company-users integration.
Actual Result	On creating a company with multiple users, the users relationship is returned as an Eloquent Collection and contains all associated users, validating the integrity and proper setup of the company-users integration.
Status	Pass
Severity	High

File: CountryTest.txt

Test ID	TC-CT-001
Title	Verify One-to-Many Relationship Between Country and Addresses
Objective	Validate that the Country model is properly integrated with the Address model such that a country record can have multiple related address records, confirming the relationship setup at the database and ORM level.
Preconditions	- Application is running in a test environment. - Database has been seeded with required data using 'DatabaseSeeder' and 'DemoSeeder'. - Laravel models for Country and Address are configured correctly. - Factories for Address are available.
Steps	Step 1: Retrieve a Country instance using Country::find(1). Step 2: Create 5 new Address records linked to that country's ID using Address::factory()->count(5)->create(). Step 3: Verify using $country->address()->exists() that the country has related addresses.
Test Data	- Country: Country with ID 1, seeded via 'DatabaseSeeder'. - Addresses: 5 new Address records created, each with country_id set to 1. - Expected: Country with ID 1 has at least one address associated.
Expected Result	Country with ID=1 successfully returns true for has addresses, confirming the existence of a one-to-many relationship between Country and Address, and the correct recording of address entries in the database.
Actual Result	Country with ID=1 successfully returns true for has addresses, confirming the existence of a one-to-many relationship between Country and Address, and the correct recording of address entries in the database.
Status	Pass
Severity	High

File: CustomFieldTest.txt

Test ID	TC-CFT-001
Title	Verify Custom Field Association with Company
Objective	Validate that a created custom field is correctly linked to a company record, ensuring the relationship is established at the database level.
Preconditions	- Application running with access to database and ORM (Eloquent). - Database seeded using DatabaseSeeder and DemoSeeder to provide required baseline data. - CustomField model and related Company model are properly configured.
Steps	Step 1: Trigger Artisan commands to seed the database, ensuring prerequisite data exists. Step 2: Create a new CustomField instance using the factory. Step 3: Check that the CustomField is linked to a company via the 'company' relationship (company()->exists()).
Test Data	- Input: New CustomField instance created via factory (default attributes). - Expected: CustomField should be associated with an existing Company (company relationship exists).
Expected Result	CustomField is successfully associated with a company; the company relationship exists and can be queried, confirming one-to-many integrity between CustomField and Company.
Actual Result	CustomField is successfully associated with a company; the company relationship exists and can be queried as expected.
Status	Pass
Severity	High

Test ID	TC-CFT-002
Title	Ensure Custom Field Maintains Multiple Custom Field Values
Objective	Validate the one-to-many relationship between a custom field and its custom field values, checking both the count and the existence of related records in the database.
Preconditions	- Application running with database and ORM access. - Database seeded using DatabaseSeeder and DemoSeeder to ensure relational integrity. - CustomField and CustomFieldValue models correctly configured and related.
Steps	Step 1: Trigger Artisan commands to seed the database with necessary data. Step 2: Create a CustomField using the factory, specifying it should have 5 custom field values. Step 3: Assert that the CustomField has exactly 5 custom field values. Step 4: Confirm that the custom field values relationship exists (customFieldValues()->exists()).
Test Data	- Input: New CustomField instance created via factory with 5 related CustomFieldValues. - Expected: CustomField must have exactly 5 related CustomFieldValues that exist in the database.
Expected Result	CustomField is associated with exactly 5 CustomFieldValues, and the relationship is properly established and can be queried for existence, confirming the integrity of the one-to-many relationship between CustomField and CustomFieldValue.
Actual Result	CustomField is associated with exactly 5 CustomFieldValues, and the relationship exists as expected.
Status	Pass
Severity	Medium

File: CustomFieldValueTest.txt

Test ID	TC-CFVT-001
Title	Verify Association Between Custom Field Value and Company
Objective	Validate that a created Custom Field Value is correctly associated with a Company entity within the integrated system, confirming relational integrity.
Preconditions	- Application environment running and accessible. - Database seeded with required core and demo data using DatabaseSeeder and DemoSeeder. - CustomFieldValue and Company models, database tables, and relationships must be configured and available.
Steps	Step 1: Seed the database with core and demo data. Step 2: Create a CustomFieldValue instance via factory. Step 3: Assert that the CustomFieldValue instance is associated with a Company (relationship exists).
Test Data	- CustomFieldValue instance created using factory (random field values). - No explicit input; association is implicit. - Expected: The company() relationship exists for the created CustomFieldValue.
Expected Result	The created CustomFieldValue instance has a valid association with a Company entity. The company() relation exists and returns true, indicating successful linkage between CustomFieldValue and Company at the database level.
Actual Result	The created CustomFieldValue instance has a valid association with a Company entity. The company() relation exists and returns true, indicating successful linkage between CustomFieldValue and Company at the database level.
Status	Pass
Severity	High

Test ID	TC-CFVT-002
Title	Verify Association Between Custom Field Value and Custom Field
Objective	Validate that a created Custom Field Value is correctly linked to a Custom Field entity within the system, ensuring relationship integrity is enforced at the database and model levels.
Preconditions	- Application environment running and accessible. - Database seeded with required core and demo data using DatabaseSeeder and DemoSeeder. - CustomFieldValue and CustomField models, corresponding database tables, and relationships must exist.
Steps	Step 1: Seed the database with core and demo data. Step 2: Create a CustomFieldValue instance associated with a CustomField via forCustomField() factory. Step 3: Assert that the CustomFieldValue instance is linked to a CustomField (relationship exists).
Test Data	- CustomFieldValue instance created with an explicit link to a CustomField using forCustomField() factory state. - No direct input; association managed through factory state. - Expected: The customField() relationship exists for the created CustomFieldValue.
Expected Result	The created CustomFieldValue instance has a valid association with a CustomField entity. The customField() relation exists and returns true, confirming correct integration between CustomFieldValue and CustomField at the model and database levels.

Actual Result	The created CustomFieldValue instance has a valid association with a CustomField entity. The customField() relation exists and returns true, confirming correct integration between CustomFieldValue and CustomField at the model and database levels.
Status	Pass
Severity	High

File: CustomerTest.txt

Test ID	TC-CT-001
Title	Verify Customer–Estimates Relationship Integration
Objective	Ensure that a Customer entity is correctly associated with multiple Estimate records, validating data integrity and relationship mappings between Customer and Estimates.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory models for Customer and Estimates available
Steps	1. Create a Customer instance with 5 associated Estimate records using factory methods. 2. Verify that the Customer object's estimates relationship contains exactly 5 items. 3. Confirm that the estimates relationship exists (i.e., there is at least one related Estimate).
Test Data	- Input: Create a Customer with 5 associated Estimates using Customer::factory()->hasEstimates(5) - Expected: Customer has 5 Estimate records; customer->estimates relationship returns 5 estimates and exists
Expected Result	Customer is successfully associated with 5 Estimate records, the relationship exists, and the estimates count matches; data integrity between Customer and Estimates is maintained.
Actual Result	Customer is successfully associated with 5 Estimate records, the relationship exists, and the estimates count matches; data integrity between Customer and Estimates is maintained.
Status	Pass
Severity	High

Test ID	TC-CT-002
Title	Validate Customer–Expenses Relationship Integration
Objective	Verify the correct association of a Customer entity with multiple Expense records, ensuring the relationship operates as intended in the integrated environment.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory models for Customer and Expenses available
Steps	1. Create a Customer instance with 5 associated Expense records via factory methods. 2. Verify the Customer object's expenses relationship contains exactly 5 items. 3. Confirm the expenses relationship exists (i.e., at least one related Expense).
Test Data	- Input: Create a Customer with 5 associated Expenses using Customer::factory()->hasExpenses(5) - Expected: Customer has 5 Expense records; customer->expenses relationship returns 5 expenses and exists
Expected Result	Customer is successfully linked to 5 Expense records, the relationship exists, and the expenses count matches; integration between Customer and Expenses is validated.
Actual Result	Customer is successfully linked to 5 Expense records, the relationship exists, and the expenses count matches; integration between Customer and Expenses is validated.
Status	Pass
Severity	Medium

Test ID	TC-CT-003
Title	Check Customer–Invoices Relationship Integration
Objective	Validate that the Customer entity correctly manages multiple Invoice relationships, testing the infrastructural connection between Customer and Invoices.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory models for Customer and Invoices available
Steps	1. Create a Customer instance with 5 associated Invoice records using appropriate factory methods. 2. Ensure Customer object's invoices relationship contains exactly 5 items. 3. Confirm the invoices relationship exists (i.e., at least one related Invoice).
Test Data	- Input: Create a Customer with 5 associated Invoices using Customer::factory()->hasInvoices(5) - Expected: Customer has 5 Invoice records; customer->invoices relationship returns 5 invoices and exists
Expected Result	Customer is correctly associated with 5 Invoice records, the relationship exists, and data accuracy in the invoices count is validated.
Actual Result	Customer is correctly associated with 5 Invoice records, the relationship exists, and data accuracy in the invoices count is validated.
Status	Pass
Severity	High

Test ID	TC-CT-004
Title	Verify Customer–Payments Relationship Mapping
Objective	Ensure integration between the Customer entity and multiple Payment records is correctly established and operational.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory models for Customer and Payments available
Steps	1. Instantiate a Customer with 5 Payment records using factory methods. 2. Confirm that the Customer object's payments relationship contains exactly 5 items. 3. Verify the payments relationship exists (i.e., at least one related Payment present).
Test Data	- Input: Create a Customer with 5 associated Payments using Customer::factory()->hasPayments(5) - Expected: Customer has 5 Payment records; customer->payments relationship returns 5 payments and exists
Expected Result	Customer is successfully mapped to 5 Payment records; payments relationship is valid and accurate.
Actual Result	Customer is successfully mapped to 5 Payment records; payments relationship is valid and accurate.
Status	Pass
Severity	High

Test ID	TC-CT-005
Title	Test Customer–Addresses Integration
Objective	Validate that a Customer can have multiple Address records and the relationship mapping is correctly maintained.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory models for Customer and Addresses available

Steps	1. Create a Customer with 5 Address records using factory methods. 2. Check that Customer object's addresses relationship includes exactly 5 Address items. 3. Verify the addresses relationship exists (i.e., at least one related Address present).
Test Data	- Input: Create a Customer with 5 associated Addresses using Customer::factory()->hasAddresses(5) - Expected: Customer has 5 Address records; customer->addresses relationship returns 5 addresses and exists
Expected Result	Customer is associated with 5 Address records, relationship exists, and addresses count accuracy is maintained.
Actual Result	Customer is associated with 5 Address records, relationship exists, and addresses count accuracy is maintained.
Status	Pass
Severity	High

Test ID	TC-CT-006
Title	Validate Customer–Currency Association
Objective	Ensure that the Customer entity is correctly linked to a Currency and the relationship exists in the integrated system.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory models for Customer and Currency available
Steps	1. Create a Customer record with default factory methods. 2. Validate that the currency relationship for the created Customer exists (at least one related Currency).
Test Data	- Input: Create a Customer using Customer::factory()->create() - Expected: The customer->currency relationship exists and points to a valid Currency record
Expected Result	Customer is successfully linked to an existing Currency; currency relationship is functional and exists.
Actual Result	Customer is successfully linked to an existing Currency; currency relationship is functional and exists.
Status	Pass
Severity	Medium

Test ID	TC-CT-007
Title	Customer–Company Relationship Integration Verification
Objective	Confirm that a Customer is correctly associated with a Company; the relationship exists and is maintained in the integrated environment.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory models for Customer and Company available
Steps	1. Create a Customer record using the factory, associating it with a Company. 2. Validate that the company relationship exists for the created Customer (at least one related Company).
Test Data	- Input: Create a Customer associated with a Company using Customer::factory()->forCompany()->create() - Expected: The customer->company relationship exists and points to a valid Company record
Expected Result	Customer is properly associated with a Company; the company relationship exists and is validated.
Actual Result	Customer is properly associated with a Company; the company relationship exists and is validated.

Status	Pass
Severity	High

Test ID	TC-CT-008
Title	Billing Address Integration for Customer Entity
Objective	Validate that a Customer can have a specific Billing Address linked, and that the billingAddress relationship works in the integrated system.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory model for Customer and Address available
Steps	1. Create a Customer with a single Address record classified as a Billing Address. 2. Validate that the billingAddress relationship exists for the created Customer.
Test Data	- Input: Create a Customer with one Address of type BILLING_TYPE using Customer::factory()->has(Address::factory()->state(['type' => Address::BILLING_TYPE])) - Expected: The customer->billingAddress relationship exists and returns the correct Address
Expected Result	Customer is successfully associated with a Billing Address; billingAddress relationship exists and is accurate.
Actual Result	Customer is successfully associated with a Billing Address; billingAddress relationship exists and is accurate.
Status	Pass
Severity	High

Test ID	TC-CT-009
Title	Shipping Address Integration for Customer Entity
Objective	Ensure that the Customer entity can be linked to a Shipping Address and that the shippingAddress relationship functions properly in the integrated system.
Preconditions	- Application environment running and accessible - Database seeded with DatabaseSeeder and DemoSeeder - Factory model for Customer and Address available
Steps	1. Create a Customer with a single Address record classified as a Shipping Address. 2. Validate that the shippingAddress relationship exists for the created Customer.
Test Data	- Input: Create a Customer with one Address of type SHIPPING_TYPE using Customer::factory()->has(Address::factory()->state(['type' => Address::SHIPPING_TYPE])) - Expected: The customer->shippingAddress relationship exists and returns the correct Address
Expected Result	Customer is successfully associated with a Shipping Address; shippingAddress relationship exists and is accurate.
Actual Result	Customer is successfully associated with a Shipping Address; shippingAddress relationship exists and is accurate.
Status	Pass
Severity	High

File: Customer\DashboardTest.txt

Test ID	TC-Cust-DT-001
Title	Customer Dashboard Endpoint Returns Success for Authenticated Customer
Objective	Validate the integration between the customer authentication system and the customer dashboard API endpoint, ensuring only properly authenticated customers can access their dashboard successfully.
Preconditions	- Application must be running and accessible. - Database must be properly seeded with default and demo data via DatabaseSeeder and DemoSeeder. - At least one customer record must exist in the database. - Sanctum authentication guard for 'customer' must be set with a valid customer session.
Steps	Step 1: Seed the database with required seeders for demo and default data. Step 2: Create a new customer using the model factory. Step 3: Authenticate as the created customer using Sanctum under the 'customer' guard. Step 4: Send a GET request to the 'api/v1/{company-slug}/customer/dashboard' endpoint as the authenticated customer. Step 5: Assert that the response status is HTTP 200 OK, indicating successful access to the dashboard.
Test Data	- Authenticated customer instance (created via Customer factory). - HTTP GET request to endpoint: api/v1/{company-slug}/customer/dashboard where {company-slug} matches the company of the authenticated customer. Expected values: HTTP 200 OK response status.
Expected Result	A properly authenticated customer receives a HTTP 200 OK status when accessing their dashboard endpoint with their company's slug, confirming authorized access to dashboard resources.
Actual Result	A properly authenticated customer receives a HTTP 200 OK status when accessing their dashboard endpoint with their company's slug, confirming authorized access to dashboard resources.
Status	Pass
Severity	High

File: Customer\EstimateTest.txt

Test ID	TC-Cust-ET-001
Title	Customer Retrieves List of Their Estimates
Objective	Validate that a customer authenticated via Sanctum can successfully fetch the list of their estimates from the API.
Preconditions	- Application running with API endpoints accessible - Database seeded with required data via DatabaseSeeder and DemoSeeder - Customer exists in the database and is authenticated using Sanctum
Steps	1. Seed the database using DatabaseSeeder and DemoSeeder 2. Create a customer using the factory and authenticate via Sanctum as that customer 3. Send a GET request to the customer estimates endpoint with proper company slug and pagination 4. Verify the HTTP response is 200 OK
Test Data	- Input: GET request to "api/v1/{company_slug}/customer/estimates?page=1" with customer authentication - Expected: HTTP 200 OK; response contains estimates for the authenticated customer
Expected Result	Customer receives a successful HTTP 200 response containing the list of their estimates.
Actual Result	Customer receives a successful HTTP 200 response containing the list of their estimates.
Status	Pass
Severity	High

Test ID	TC-Cust-ET-002
Title	Customer Retrieves a Specific Estimate
Objective	Validate that a customer can fetch the details of a specific estimate assigned to them through the API.
Preconditions	- Application running and API accessible - Database seeded (DatabaseSeeder and DemoSeeder) - Customer exists and is authenticated via Sanctum - An estimate is present in the database tied to the customer
Steps	1. Seed the database 2. Create and authenticate a customer 3. Create an estimate linked to the customer via factory 4. Send a GET request to retrieve details of the specific estimate 5. Check the HTTP response is 200 OK
Test Data	- Input: GET request to "api/v1/{company_slug}/customer/estimates/{estimate_id}" with customer authentication - Expected: HTTP 200 OK; response contains details of the specified estimate
Expected Result	Customer receives a 200 OK HTTP response containing the details of the specific estimate assigned to them.
Actual Result	Customer receives a 200 OK HTTP response containing the details of the specific estimate assigned to them.
Status	Pass
Severity	High

Test ID	TC-Cust-ET-003
Title	Customer Marks Estimate as Accepted
Objective	Ensure that a customer can use the API to update an estimate’s status to 'accepted' and that the change is persisted.

Preconditions	- Application and API operational - Database seeded with relevant data and demo entities - Authenticated customer exists - Estimate exists and is associated to the authenticated customer
Steps	1. Seed the database 2. Create and authenticate a customer 3. Generate an estimate for the customer with appropriate dates 4. Send a POST request to update the estimate status to 'accepted' 5. Verify the HTTP response is 200 OK and the status field in the response equals 'accepted'
Test Data	- Input: POST request to "api/v1/{company_slug}/customer/estimate/{estimate_id}/status" with payload: {status: STATUS_ACCEPTED} - Expected: HTTP 200 OK; response data indicates estimate status is set to 'accepted'
Expected Result	Customer receives a HTTP 200 OK response and the estimate status in the response data is updated to 'accepted'.
Actual Result	Customer receives a HTTP 200 OK response and the estimate status in the response data is updated to 'accepted'.
Status	Pass
Severity	High

Test ID	TC-Cust-ET-004
Title	Customer Marks Estimate as Rejected
Objective	Verify that a customer can reject an estimate via the API and that the status update is reflected in the backend.
Preconditions	- API endpoints available and database seeded - Authenticated customer present - Estimate available for the customer
Steps	1. Seed all necessary data in the database 2. Create a customer and authenticate them with Sanctum 3. Create an estimate for the customer 4. Send a POST request with status 'rejected' to the estimate status endpoint 5. Assert HTTP response is 200 OK and status field in response data equals 'rejected'
Test Data	- Input: POST request to "api/v1/{company_slug}/customer/estimate/{estimate_id}/status" with payload: {status: STATUS_REJECTED} - Expected: HTTP 200 OK; response data shows estimate status updated to 'rejected'
Expected Result	Customer receives a HTTP 200 OK response and the estimate status in the response data is updated to 'rejected'.
Actual Result	Customer receives a HTTP 200 OK response and the estimate status in the response data is updated to 'rejected'.
Status	Pass
Severity	High

File: Customer\ExpenseTest.txt

Test ID	TC-Cust-ET-001
Title	Retrieve List of Expenses for Authenticated Customer
Objective	Validate that the API correctly retrieves a paginated list of expenses for the currently authenticated customer, confirming integration between authentication, routing, and the expenses data source.
Preconditions	- Application server is running and accessible. - Database has been seeded with initial data (DatabaseSeeder and DemoSeeder). - At least one customer record exists in the database. - Customer is authenticated via Sanctum as per test setup.
Steps	Step 1: Authenticate as a seeded customer using Sanctum. Step 2: Send a GET request to fetch the expenses list for the customer. Step 3: Verify the response returns HTTP 200 OK and includes the correct expenses data structure.
Test Data	- Authenticated customer credentials (via Sanctum). - GET request to endpoint: api/v1/{company_slug}/customer/expenses?page=1 - No specific body payload; relies on authenticated session and seeded data. Expected values: - HTTP 200 OK response. - JSON containing paginated list of expenses pertaining to the authenticated customer.
Expected Result	API responds with HTTP 200 OK and returns a paginated JSON list of expenses that belong to the authenticated customer.
Actual Result	API responds with HTTP 200 OK and returns a paginated JSON list of expenses that belong to the authenticated customer.
Status	Pass
Severity	High

Test ID	TC-Cust-ET-002
Title	Retrieve Individual Expense Details for Authenticated Customer
Objective	Ensure that the API allows an authenticated customer to access detailed information about an individual expense, correctly enforcing access controls and data retrieval from the database.
Preconditions	- Application server is running and accessible. - Database has been seeded with initial data (DatabaseSeeder and DemoSeeder). - At least one expense exists linked to the authenticated customer. - Customer is authenticated via Sanctum as per test setup.
Steps	Step 1: Authenticate as a seeded customer using Sanctum. Step 2: Create a new expense in the database linked to logged-in customer and company. Step 3: Send a GET request to fetch the expense details for the created expense. Step 4: Verify the response returns HTTP 200 OK and contains the expected expense data.
Test Data	- Authenticated customer credentials (via Sanctum). - GET request to endpoint: /api/v1/{company_slug}/customer/expenses/{expense_id} - Expense record created for the specific customer and company in test setup. Expected values: - HTTP 200 OK response. - JSON containing the details of the requested expense record.

Expected Result	API responds with HTTP 200 OK and returns JSON containing detailed information for the requested expense, provided it belongs to the authenticated customer.
Actual Result	API responds with HTTP 200 OK and returns JSON containing detailed information for the requested expense, provided it belongs to the authenticated customer.
Status	Pass
Severity	High

File: Customer\InvoiceTest.txt

Test ID	TC-Cust-IT-001
Title	Retrieve paginated list of invoices for an authenticated customer
Objective	Validate the integration between customer authentication, API routing, and invoice retrieval endpoint; ensure an authenticated customer receives their invoice list.
Preconditions	- Application server is running with necessary configurations. - Database is seeded with sample companies, customers, and invoice data using DatabaseSeeder and DemoSeeder. - Customer is authenticated using Laravel Sanctum as 'customer' guard. - API endpoint for customer invoices is accessible.
Steps	Step 1: Ensure the database is seeded and a customer is authenticated via Sanctum. Step 2: Send a GET request to the customer invoices API endpoint with valid company slug and page parameter. Step 3: Validate the response status is HTTP 200 OK and the payload contains a list of invoices.
Test Data	- Authenticated Customer (created via Customer factory). - GET request to: api/v1/{company_slug}/customer/invoices?page=1 Expected values: HTTP 200 OK (success), JSON payload containing one or more invoices, belonging to authenticated customer.
Expected Result	HTTP 200 OK is returned; JSON response contains invoices associated with the authenticated customer, confirming access control and data integration between authentication and invoice models.
Actual Result	HTTP 200 OK is returned; JSON response contains invoices associated with the authenticated customer.
Status	Pass
Severity	High

Test ID	TC-Cust-IT-002
Title	Retrieve specific invoice details for authenticated customer and verify database entry
Objective	Verify the integration between customer authentication, invoice creation, and retrieval endpoints; ensure invoice details returned by the API match database records for the customer.
Preconditions	- Application server running with required configuration. - Database seeded using DatabaseSeeder and DemoSeeder. - Authenticated customer via Laravel Sanctum ('customer' guard). - API endpoint for retrieving individual invoices is accessible.
Steps	Step 1: Seed the database and authenticate customer. Step 2: Create an invoice linked to the authenticated customer in the database. Step 3: Send a GET request to the customer invoice detail endpoint (with company slug and invoice ID). Step 4: Validate response status is HTTP 200 OK. Step 5: Confirm invoice exists in the database with expected attribute values matching those returned by the API.
Test Data	- Authenticated Customer (created using factory). - Invoice instance created with 'customer_id' set to authenticated customer. - GET request to: /api/v1/{company_slug}/customer/invoices/{invoice_id} Expected values: HTTP 200 OK (success), JSON payload with invoice details matching database values including template_name, invoice_number, sub_total, discount, customer_id, total, tax.
Expected Result	HTTP 200 OK returned; JSON response contains all invoice details for the specified invoice belonging to the authenticated customer; database contains matching invoice record with correct attributes.

Actual Result	HTTP 200 OK returned; JSON response contains all invoice details for the specified invoice belonging to the authenticated customer; database contains matching invoice record with correct attributes.
Status	Pass
Severity	High

File: Customer\PaymentTest.txt

Test ID	TC-Cust-PT-001
Title	Retrieve All Customer Payments via API
Objective	Validate that the API can successfully return a paginated list of payments associated with an authenticated customer, ensuring integration between authentication, API endpoint, and underlying database.
Preconditions	- Application server is running in test or staging environment - Database is seeded using DatabaseSeeder and DemoSeeder - At least one customer exists in the database (created via factory) - API authentication is active via Sanctum; customer is acting as current user
Steps	Step 1: Seed the database and set up a customer using factories. Step 2: Authenticate the customer session using Sanctum. Step 3: Send GET request to API endpoint for the customer's payments with page=1. Step 4: Validate that the response status is HTTP 200 OK.
Test Data	- Authenticated customer credentials (via Sanctum) - GET request to: api/v1/{company-slug}/customer/payments?page=1 - Expected HTTP response status: 200 OK
Expected Result	API returns an HTTP 200 OK status, indicating the list of customer payments is successfully retrieved, confirming integration among authentication, routing, and database.
Actual Result	API returns an HTTP 200 OK status, indicating the list of customer payments is successfully retrieved.
Status	Pass
Severity	High

Test ID	TC-Cust-PT-002
Title	Retrieve Specific Customer Payment via API
Objective	Validate that the API can fetch details for a specific payment belonging to the authenticated customer, verifying correct integration between payment records, user authentication, and API routing.
Preconditions	- Application server is running in test or staging environment - Database is seeded using DatabaseSeeder and DemoSeeder - At least one customer exists in the database (created via factory) - At least one payment exists and is assigned to the authenticated customer - API authentication is active via Sanctum; customer is acting as current user
Steps	Step 1: Seed the database and create a customer using factories. Step 2: Authenticate the customer session using Sanctum. Step 3: Create a payment record in the database assigned to this customer. Step 4: Send GET request to API endpoint for that specific payment record. Step 5: Validate that the response status is HTTP 200 OK.
Test Data	- Authenticated customer credentials (via Sanctum) - Payment record with 'customer_id' matching current user - GET request to: /api/v1/{company-slug}/customer/payments/{payment-id} - Expected HTTP response status: 200 OK
Expected Result	API returns an HTTP 200 OK status, confirming that the payment exists for the authenticated customer and the detail is retrievable, ensuring proper integration between authentication, database, and API routing.
Actual Result	API returns an HTTP 200 OK status, confirming that the payment exists for the authenticated customer.
Status	Pass
Severity	High

File: Customer\ProfileTest.txt

Test ID	TC-Cust-PT-001
Title	Verify Customer Profile Update Employs Form Request Validation
Objective	To confirm that updating a customer profile triggers validation logic through the designated form request, ensuring endpoint and request are integrated appropriately.
Preconditions	- Application server is running. - Database is seeded with standard and demo data. - Authenticated Customer session via Sanctum. - ProfileController and CustomerProfileRequest are implemented and available.
Steps	Step 1: Attempt to initiate the updateProfile action in ProfileController. Step 2: Verify that the updateProfile action uses CustomerProfileRequest for request validation.
Test Data	- Controller: ProfileController::updateProfile - Form request: CustomerProfileRequest - No specific customer data since test is at controller/form-request linkage level.
Expected Result	The updateProfile action in ProfileController requires and utilizes CustomerProfileRequest for input validation, ensuring integration between controller and form validation is enforced for customer profile updates.
Actual Result	The updateProfile action in ProfileController requires and utilizes CustomerProfileRequest for input validation, ensuring integration between controller and form validation is enforced for customer profile updates.
Status	Pass
Severity	High

Test ID	TC-Cust-PT-002
Title	Update Customer Profile via API and Verify Persistence to Database
Objective	To validate the integration between the Customer profile update API endpoint and the underlying database by ensuring updates are accepted and correctly persisted.
Preconditions	- Application server is running. - Database is seeded with valid customer and demo data. - Authenticated Customer session via Sanctum. - Customer exists in DB related to the company slug.
Steps	Step 1: Retrieve current authenticated customer instance. Step 2: Prepare new profile data using Customer factory with updated shipping and billing info. Step 3: Send POST request to customer profile API endpoint with new data. Step 4: Assert HTTP 200 OK response confirming successful update. Step 5: Validate via database assertion that customer's name and email persist in 'customers' table (i.e., no deletion, correct update).

Test Data	- API Endpoint: POST api/v1/{company_slug}/customer/profile
	- Request payload:
	shipping: {
	name: 'newName',
	address_street_1: 'address'
	},
	billing: {
	name: 'newName',
	address_street_1: 'address'
	}
	- Customer identifying attributes (name, email) for DB validation.

Expected Result	The API response returns HTTP 200 OK indicating success. The updated customer profile data is integrated and persisted into the database, and the 'customers' table contains the expected name and email for the updated customer.
Actual Result	The API response returns HTTP 200 OK indicating success. The updated customer profile data is integrated and persisted into the database, and the 'customers' table contains the expected name and email for the updated customer.
Status	Pass
Severity	High

Test ID	TC-Cust-PT-003
Title	Retrieve Customer Profile via Authenticated API Request
Objective	To verify the end-to-end integration between the customer authentication mechanism and the profile retrieval API, ensuring that authenticated customers can access their profile details.
Preconditions	- Application server is running. - Database is seeded with standard and demo data. - Customer authenticated using Sanctum. - Existing customer linked to company slug.
Steps	Step 1: Retrieve current authenticated customer instance. Step 2: Send GET request to customer self-lookup API endpoint. Step 3: Confirm HTTP 200 OK response indicating successful retrieval.
Test Data	- API Endpoint: GET api/v1/{company_slug}/customer/me - Authenticated customer session context.
Expected Result	The GET request to the customer profile endpoint returns HTTP 200 OK. The system correctly retrieves and responds with the customer's profile data, confirming integration between authentication and data retrieval functionality.
Actual Result	The GET request to the customer profile endpoint returns HTTP 200 OK. The system correctly retrieves and responds with the customer's profile data, confirming integration between authentication and data retrieval functionality.
Status	Pass
Severity	High

File: EstimateItemTest.txt

Test ID	TC-EIT-001
Title	Verify EstimateItem Association with Estimate
Objective	Ensure that when an EstimateItem is created and linked to an Estimate, the relationship between EstimateItem and Estimate is correctly persisted and accessible in the system.
Preconditions	- Application server running with access to database - Database seeded with test data using DatabaseSeeder and DemoSeeder - Models and relationships are defined and enabled
Steps	Step 1: Seed the database with initial data. Step 2: Create an EstimateItem factory instance associated with an Estimate. Step 3: Query the Estimate relationship from EstimateItem. Step 4: Assert that the Estimate relationship exists.
Test Data	- EstimateItem created via factory with associated Estimate (forEstimate()) - No specific values required beyond valid foreign key relationship
Expected Result	EstimateItem correctly references an existing Estimate; querying the relationship returns a valid, existing Estimate.
Actual Result	EstimateItem correctly references an existing Estimate; querying the relationship returns a valid, existing Estimate.
Status	Pass
Severity	High

Test ID	TC-EIT-002
Title	Validate EstimateItem Association with Item
Objective	Verify that the EstimateItem model is correctly associated with an Item, and the relationship between EstimateItem and Item functions as expected within the system.
Preconditions	- Application server running with database access - Database seeded with required data using seeders - Estimate and Item models and relationships available
Steps	Step 1: Seed database with prerequisites. Step 2: Create an EstimateItem with linked Item and Estimate via factories. Step 3: Query the Item relationship from EstimateItem. Step 4: Assert that the Item relationship exists.
Test Data	- EstimateItem created via factory with associated Item and Estimate (using factories) - Valid item_id and estimate_id values used
Expected Result	EstimateItem correctly references an existing Item; querying the Item relationship returns a valid, existing Item.
Actual Result	EstimateItem correctly references an existing Item; querying the Item relationship returns a valid, existing Item.
Status	Pass
Severity	High

Test ID	TC-EIT-003
Title	Validate EstimateItem Multi-Tax Relationship
Objective	Ensure that an EstimateItem can be associated with multiple Tax entities, and the system correctly persists and retrieves all associated taxes.
Preconditions	- Application running and accessible - Database seeded with required Estimate and supporting data - Taxes relationship on EstimateItem model configured

Steps	Step 1: Seed the database with necessary data. Step 2: Create an EstimateItem via factory with five associated taxes and a related Estimate. Step 3: Assert that EstimateItem has exactly five Tax records. Step 4: Assert that the taxes relationship exists and returns records.
Test Data	- EstimateItem created via factory with hasTaxes(5) and associated Estimate - Five related Tax records expected
Expected Result	EstimateItem is associated with exactly five Tax records; querying taxes returns a collection of five valid taxes and the relationship exists.
Actual Result	EstimateItem is associated with exactly five Tax records; querying taxes returns a collection of five valid taxes and the relationship exists.
Status	Pass
Severity	High

File: EstimateTest.txt

Test ID	TC-ET-001
Title	Verify Estimate Entity Links Multiple Estimate Items
Objective	To validate that an Estimate model correctly establishes and retrieves associations with multiple Estimate Items in the database.
Preconditions	- Application server running with full Laravel environment - Database seeded with required baseline and demo data via DatabaseSeeder and DemoSeeder - Factories for Estimate and EstimateItem available and functional
Steps	1. Create an Estimate instance using a factory, specifying the presence of 5 associated items. 2. Retrieve Estimate Items related to the created Estimate instance. 3. Assert that the Estimate has exactly 5 items associated. 4. Verify that the relationship returns true for the existence of items.
Test Data	- An Estimate instance created with 5 associated Estimate Items (factory-generated) - No direct input data used, only model factories
Expected Result	The Estimate model should have 5 linked Estimate Items, and its items relationship should be populated and accessible.
Actual Result	The Estimate model has 5 linked Estimate Items, and its items relationship is populated and accessible.
Status	Pass
Severity	High

Test ID	TC-ET-002
Title	Confirm Estimate Entity is Properly Linked to a Customer
Objective	To ensure that the business logic correctly associates each Estimate with a single Customer entity.
Preconditions	- Application and database up and seeded with customers - Factories for Estimate and Customer available
Steps	1. Create an Estimate instance with an associated Customer using model factories. 2. Assert that the customer relationship on the Estimate returns true for existence.
Test Data	- An Estimate created with an associated Customer (using forCustomer factory method)
Expected Result	Each Estimate instance should have a valid, retrievable Customer association.
Actual Result	The Estimate instance has a valid, retrievable Customer association.
Status	Pass
Severity	High

Test ID	TC-ET-003
Title	Validate Multiple Tax Associations for an Estimate
Objective	To verify that an Estimate model can correctly link and retrieve several Tax entities.
Preconditions	- Application running with demo data seeded - Factories for Estimate and Tax present
Steps	1. Create an Estimate with 5 linked Taxes. 2. Retrieve the Taxes related to the Estimate and count them. 3. Assert that the number of taxes is 5. 4. Ensure the taxes relationship on the Estimate returns true for existence.

Test Data	- An Estimate instance created with 5 associated Taxes (factory-generated)
Expected Result	The Estimate entity must have 5 associated Tax records, and its taxes relationship must be correctly populated.
Actual Result	The Estimate entity has 5 associated Tax records and its taxes relationship is populated.
Status	Pass
Severity	High

Test ID	TC-ET-004
Title	Create a New Estimate with Items and Taxes Persisted in Database
Objective	To verify that the Estimate creation process correctly saves Estimate data, associated Items, and Taxes to the database via an incoming request.
Preconditions	- Laravel application running with seeded database - Factories for Estimate, EstimateItem, and Tax in place - Estimate creation API available
Steps	1. Prepare a raw Estimate payload, including one raw Item and one raw Tax. 2. Construct a new EstimatesRequest, replacing its data with the prepared estimate and related objects. 3. Call Estimate::createEstimate() with this request. 4. Assert the created EstimateItem exists in the database and matches the input values. 5. Assert the created Estimate exists in the database and matches the input values.
Test Data	- Estimate request payload (factory raw data) - One Estimate Item and one Tax (factory raw data) included in the request
Expected Result	An Estimate, its associated Item, and Tax are all created and persisted to the database with correct values as provided in the request.
Actual Result	An Estimate, its associated Item, and Tax are all created and persisted to the database with correct values as provided in the request.
Status	Pass
Severity	High

Test ID	TC-ET-005
Title	Update Existing Estimate and Synchronize Related Items and Taxes
Objective	To validate that updating an Estimate properly updates the Estimate record, its Items, and Taxes in the database as per the modified payload.
Preconditions	- Application server running with database seeded (DatabaseSeeder, DemoSeeder) - Existing Estimate with linked Items and Taxes
Steps	1. Create an Estimate with associated Items and Taxes. 2. Prepare new raw Estimate data and an updated Item and Tax. 3. Replace existing Estimate with new values via an EstimatesRequest object. 4. Call updateEstimate on the Estimate instance. 5. Assert the updated EstimateItem exists with new details. 6. Assert the updated Estimate exists with all new attributes.
Test Data	- Existing Estimate with current Items and Taxes - New request payload: updated Estimate data, updated Item, and updated Tax
Expected Result	The existing Estimate record, its Items, and Taxes are all updated in the database, and their attributes match the updated input values.
Actual Result	The existing Estimate record, its Items, and Taxes are all updated in the database, and their attributes match the updated input values.
Status	Pass

Severity	High

Test ID	TC-ET-006
Title	Create and Persist Estimate Items to Database
Objective	To confirm that the Estimate::createItems method correctly saves provided Items against an existing Estimate.
Preconditions	- Application and database seeded correctly - Existing Estimate present
Steps	1. Create an Estimate instance. 2. Prepare one Item linked to the Estimate (invoice_id set). 3. Send a Request containing the Item list to Estimate::createItems. 4. Assert the Item is persisted in the database and its values match input. 5. Assert the Estimate model has exactly 1 Item.
Test Data	- Estimate instance (factory-generated) - One EstimateItem (factory raw data; associated to the Estimate via invoice_id)
Expected Result	The Item is created in the database under the corresponding Estimate, with all values matched and the Estimate’s items count accurately updated.
Actual Result	The Item is created in the database under the corresponding Estimate, with all values matched and the Estimate’s items count accurately updated.
Status	Pass
Severity	Medium

Test ID	TC-ET-007
Title	Create and Persist Taxes for an Estimate
Objective	To ensure that Taxes associated with an Estimate are correctly created and saved to the database through Estimate::createTaxes.
Preconditions	- Application and database seeded via DatabaseSeeder and DemoSeeder - Estimate instance present
Steps	1. Create an Estimate instance. 2. Prepare two raw Tax objects, linked to the Estimate. 3. Create a Request containing the tax list. 4. Call Estimate::createTaxes with the Estimate and the Request. 5. Assert the Estimate model has 2 Taxes linked. 6. Assert each Tax record is present in the database and matches the provided input.
Test Data	- Estimate instance (factory-generated) - Two Tax records (factory raw data, estimate_id set)
Expected Result	Both Tax records are successfully persisted in the database, accurately linked to the Estimate, with the details matching input values.
Actual Result	Both Tax records are successfully persisted in the database, accurately linked to the Estimate, with the details matching input values.
Status	Pass
Severity	Medium

File: ExchangeRateLogTest.txt

Test ID	TC-ERLT-001
Title	Verify that an exchange rate log is correctly associated with a company
Objective	Ensure that the ExchangeRateLog entry created is linked to a valid company record, confirming the relationship between exchange rate logs and companies in the system.
Preconditions	- Application is running in a test environment with database access. - Database is seeded using DatabaseSeeder and DemoSeeder for minimum data setup. - Factory definitions for ExchangeRateLog and Company exist.
Steps	Step 1: Create an ExchangeRateLog using a factory with the forCompany() method to link it to a company. Step 2: Check if the associated company of the ExchangeRateLog exists in the database.
Test Data	- No explicit input data; ExchangeRateLog is generated via factory with company association. - Expected value: The created ExchangeRateLog has an associated Company that exists.
Expected Result	The ExchangeRateLog entry is successfully associated with a company record, and the related company exists in the database.
Actual Result	The ExchangeRateLog entry is successfully associated with a company record, and the related company exists in the database.
Status	Pass
Severity	High

Test ID	TC-ERLT-002
Title	Validate creation and database entry of an exchange rate log when adding a new expense
Objective	Ensure that when a new expense is created, an exchange rate log is generated and correctly inserted into the database with valid fields reflecting the transaction.
Preconditions	- Application is running in a test environment with database access. - Database is seeded using DatabaseSeeder and DemoSeeder for base and demo data. - Factory definitions for Expense and ExchangeRateLog exist.
Steps	Step 1: Create an Expense record using factory. Step 2: Use ExchangeRateLog::addExchangeRateLog() to add an exchange rate log linked to the expense. Step 3: Assert that the exchange_Rate_logs table contains the new exchange rate log with the fields: exchange_rate, base_currency_id, and currency_id matching the response.
Test Data	- Input: Expense record generated via factory. - Expected: Corresponding ExchangeRateLog with correct exchange_rate, base_currency_id, and currency_id values reflecting the expense.
Expected Result	A new exchange rate log entry is present in the exchange_Rate_logs table, correctly populated with exchange_rate, base_currency_id, and currency_id values matching the related expense transaction.
Actual Result	A new exchange rate log entry is present in the exchange_Rate_logs table, correctly populated with exchange_rate, base_currency_id, and currency_id values matching the related expense transaction.
Status	Pass
Severity	High

File: ExpenseCategoryTest.txt

Test ID	TC-ECT-001
Title	Verify ExpenseCategory-to-Expenses Relationship Mapping
Objective	Validate that an ExpenseCategory correctly maintains its relationship with multiple Expense records upon creation, confirming integrity of the expense-categories linkage in the ORM and database.
Preconditions	- Application with required dependencies up and running - Database seeded with DatabaseSeeder and DemoSeeder - Laravel Eloquent models and factories available - ExpenseCategory and related Expense tables exist
Steps	Step 1: Seed database via DatabaseSeeder and DemoSeeder to ensure baseline data and relations Step 2: Use ExpenseCategory factory to create a category with 5 associated Expense entries Step 3: Assert that the created ExpenseCategory has exactly 5 Expense records linked (count verification) Step 4: Assert that the expenses() relationship exists and is retrievable for this category
Test Data	- ExpenseCategory created using factory, with 5 linked Expense records via hasExpenses(5) - Expected values: - ExpenseCategory has exactly 5 related Expense records - Expenses relationship exists and is retrievable
Expected Result	ExpenseCategory created via the factory has exactly 5 associated Expense records; the relationship is correctly established in the ORM and database, and expenses() relationship returns true for existence.
Actual Result	ExpenseCategory created via the factory has exactly 5 associated Expense records; the relationship is correctly established in the ORM and database, and expenses() relationship returns true for existence.
Status	Pass
Severity	High

File: ExpenseTest.txt

Test ID	TC-ET-001
Title	Verify Expense Association with Category
Objective	Validate that an Expense is correctly linked to its Category in the system, ensuring the relationship between Expense and Category is operational at the data layer.
Preconditions	- Application is running with a functioning database. - Database seeded with demo and core data (via DatabaseSeeder and DemoSeeder). - Expense categories are available in the database.
Steps	Step 1: Seed the database with required core and demo data. Step 2: Create an Expense instance using the factory, specifically linking it with a Category via `forCategory()`. Step 3: Verify that the Category relationship exists on the Expense object using the relationship existence check.
Test Data	- Expense created via factory with `forCategory()` modifier. - Expected: Expense record has an associated Category; Category relationship exists for the new Expense.
Expected Result	Expense is created and has an existing association with a Category; the Category relationship can be retrieved and exists in the database.
Actual Result	Expense is created and has an existing association with a Category; the Category relationship exists as expected.
Status	Pass
Severity	High

Test ID	TC-ET-002
Title	Verify Expense Association with Customer
Objective	Validate that an Expense is correctly linked to a Customer, ensuring integrated data consistency between Expenses and Customers.
Preconditions	- Application is running with seeded demo and core data. - Database contains Customer records. - Expense factory is available and functional.
Steps	Step 1: Seed the database with core and demo data. Step 2: Create an Expense instance linked to a Customer via the factory's `forCustomer()` modifier. Step 3: Assert that the Customer relationship exists for the Expense object.
Test Data	- Expense created using factory `forCustomer()` modifier. - Expected: Expense has a valid Customer relationship in the database.
Expected Result	Expense record successfully linked to a Customer; Customer relationship exists and can be retrieved.
Actual Result	Expense record successfully linked to a Customer; Customer relationship exists.
Status	Pass
Severity	High

Test ID	TC-ET-003
Title	Verify Expense Association with Company
Objective	Ensure that Expenses are correctly attached to a Company in the database, validating integration between the Expense and Company entities.
Preconditions	- Application running with database seeded (core and demo data). - Company records exist in the system. - Expense factory available for testing.

Steps	Step 1: Seed the database with required core and demo data. Step 2: Create an Expense instance linked to a Company using the factory's `forCompany()` modifier. Step 3: Verify existence of the Company relationship for the Expense object.
Test Data	- Expense created using factory `forCompany()` modifier. - Expected: Expense object has a valid Company relationship in the database.
Expected Result	Expense is successfully linked to a Company; the Company relationship exists and is retrievable from the database.
Actual Result	Expense is successfully linked to a Company; the Company relationship exists as required.
Status	Pass
Severity	High

File: InvoiceItemTest.txt

Test ID	TC-IIT-001
Title	Verify Invoice Item Association with Invoice
Objective	Validate that when an invoice item is created, it is correctly associated with an invoice in the system, ensuring the relational integrity between invoice items and invoices.
Preconditions	Application running with access to the necessary database Database seeded with core and demo data using DatabaseSeeder and DemoSeeder Invoice and InvoiceItem models available
Steps	Step 1: Seed the database with required data via seeders Step 2: Create a new InvoiceItem using the factory, linked to a new Invoice Step 3: Check that the InvoiceItem's invoice relationship exists (invoice() returns a related invoice)
Test Data	Input: New InvoiceItem created using factory, linked to a generated Invoice via forInvoice() method Expected: The created InvoiceItem's invoice relationship should exist in the database
Expected Result	InvoiceItem is linked to a valid, existing Invoice in the database; invoice relationship is present and exists method returns true.
Actual Result	InvoiceItem is linked to a valid, existing Invoice in the database; invoice relationship is present and exists method returns true.
Status	Pass
Severity	High

Test ID	TC-IIT-002
Title	Verify Invoice Item Association with Product Item
Objective	Ensure that invoice items created in the system are correctly linked to the corresponding product items, maintaining integrity between invoice details and defined items.
Preconditions	Application running Database seeded with main and demo data (DatabaseSeeder, DemoSeeder) Item and Invoice models accessible
Steps	Step 1: Seed the database with necessary tables and sample data Step 2: Create an InvoiceItem using the factory, assigning it to a newly created Item and Invoice Step 3: Verify that the InvoiceItem's item relation exists in the database (item() returns association)
Test Data	Input: InvoiceItem created via factory, explicitly associated with new Item and Invoice Expected: The created InvoiceItem should have a valid item relationship in the database
Expected Result	InvoiceItem is correctly associated with an existing Item; item relationship exists and is retrievable from the database.
Actual Result	InvoiceItem is correctly associated with an existing Item; item relationship exists and is retrievable from the database.
Status	Pass
Severity	High

Test ID	TC-IIT-003
Title	Verify Invoice Item Association with Multiple Taxes
Objective	Confirm that an invoice item can be associated with multiple tax records, and that these associations are correctly created and reflected in the system.

Preconditions	Application running with database connection Database seeded via DatabaseSeeder and DemoSeeder Tax, Invoice, and InvoiceItem models available
Steps	Step 1: Initialize database state via seeders Step 2: Create an InvoiceItem with five associated taxes, connected to a new Invoice Step 3: Verify the count of taxes linked to InvoiceItem is exactly five Step 4: Confirm that the taxes() relationship exists for the InvoiceItem
Test Data	Input: InvoiceItem created with factory, linked to a new Invoice, with five tax records attached using hasTaxes(5) Expected: The InvoiceItem has exactly five taxes associated and the relationship exists in the database
Expected Result	InvoiceItem is associated with five distinct tax records; taxes relationship exists and count of taxes is precisely five.
Actual Result	InvoiceItem is associated with five distinct tax records; taxes relationship exists and count of taxes is precisely five.
Status	Pass
Severity	Medium

File: InvoiceTest.txt

Test ID	TC-IT-001
Title	Verify Invoice Contains Associated Invoice Items
Objective	Validate that an Invoice correctly retrieves multiple associated InvoiceItem entities from the database, confirming the ORM relationships.
Preconditions	- Application environment running. - Database seeded with required entities via DatabaseSeeder and DemoSeeder.
Steps	1. Create an Invoice with 5 associated InvoiceItems using a factory. 2. Retrieve the items relationship from the Invoice. 3. Assert the count of items is exactly 5. 4. Assert the items relationship exists in the database.
Test Data	- Invoice created using factory, configured with 5 associated InvoiceItems. - Assertions check the number of items and their existence.
Expected Result	The Invoice instance returns exactly 5 associated InvoiceItem entities, and the relationship between them exists in the database.
Actual Result	The Invoice instance returns exactly 5 associated InvoiceItem entities, and the relationship between them exists in the database.
Status	Pass
Severity	High

Test ID	TC-IT-002
Title	Verify Invoice Contains Associated Taxes
Objective	Ensure that an Invoice can correctly retrieve multiple Tax entities, verifying the many-to-many relationship and data consistency.
Preconditions	- Application server and API running. - Database seeded via DatabaseSeeder and DemoSeeder.
Steps	1. Create an Invoice with 5 associated Taxes using a factory. 2. Retrieve the taxes relationship from the Invoice. 3. Assert exactly 5 Taxes are present. 4. Assert the relationship exists in the database.
Test Data	- Invoice created with 5 associated Taxes using a factory.
Expected Result	The Invoice contains exactly 5 associated Tax entities, and the relationship exists in the database.
Actual Result	The Invoice contains exactly 5 associated Tax entities, and the relationship exists in the database.
Status	Pass
Severity	Medium

Test ID	TC-IT-003
Title	Verify Invoice Contains Associated Payments
Objective	Ensure that an Invoice correctly retrieves all associated Payment entities, confirming correct linkage and ORM relationship.
Preconditions	- Application running. - Database seeded via required seeders.
Steps	1. Create an Invoice with 5 associated Payments using a factory. 2. Fetch the payments relationship from the Invoice. 3. Assert exactly 5 Payments are present. 4. Assert the payments relationship exists in the database.
Test Data	- Invoice created with 5 associated Payments via factory configuration.

Expected Result	The Invoice instance lists 5 Payment entities and the relationship is present in the database.
Actual Result	The Invoice instance lists 5 Payment entities and the relationship is present in the database.
Status	Pass
Severity	Medium

Test ID	TC-IT-004
Title	Verify Invoice Belongs to a Customer
Objective	Check that an Invoice correctly links to a Customer entity, validating the "belongs to" relationship on the model.
Preconditions	- Application and database instance running. - Database seeded with Customers and Invoices.
Steps	1. Create an Invoice using factory configured to link to a Customer. 2. Verify that the customer relationship exists using the Invoice model relationship.
Test Data	- Invoice created using a factory and linked to a Customer.
Expected Result	The Invoice links to a Customer, and the relationship exists in the database.
Actual Result	The Invoice links to a Customer, and the relationship exists in the database.
Status	Pass
Severity	High

Test ID	TC-IT-005
Title	Verify Invoice Returns Previous Status
Objective	Ensure that an Invoice can return its previous status, confirming accurate status transition handling.
Preconditions	- Application running. - Database seeded for invoices.
Steps	1. Create an Invoice using a factory. 2. Request the previous status from the Invoice. 3. Assert the returned status equals 'DRAFT'.
Test Data	- Invoice created using factory. - Expected previous status returned: 'DRAFT'.
Expected Result	The previous status for a newly created Invoice is returned as 'DRAFT'.
Actual Result	The previous status for a newly created Invoice is returned as 'DRAFT'.
Status	Pass
Severity	Medium

Test ID	TC-IT-006
Title	Verify Invoice Creation with Items and Taxes
Objective	Validate system-level creation of an Invoice that includes line items and tax information, ensuring proper persistence in the database via integrated request handling.
Preconditions	- Application and API endpoints running. - Database seeded with required customers, templates, etc.
Steps	1. Generate raw Invoice data using the factory. 2. Generate raw InvoiceItem and raw Tax data. 3. Populate the invoice request payload with items and taxes. 4. Send the payload to the Invoice::createInvoice method. 5. Assert that the created Invoice and InvoiceItem are present in the database with the correct data.

Test Data	- Raw Invoice data including attributes (invoice_number, sub_total, total, tax, discount, notes, template_name, customer_id). - One raw InvoiceItem. - One raw Tax.
Expected Result	The Invoice, its associated InvoiceItem, and Tax are persisted in the database, with all supplied data accurately saved.
Actual Result	The Invoice, its associated InvoiceItem, and Tax are persisted in the database, with all supplied data accurately saved.
Status	Pass
Severity	High

Test ID	TC-IT-007
Title	Verify Invoice Update with New Items and Taxes
Objective	Validate system-level update of an existing Invoice, including new line items and taxes, ensuring correct database update and data integrity.
Preconditions	- Application and API running. - Database contains existing Invoice records.
Steps	1. Create an existing Invoice. 2. Generate new raw Invoice data. 3. Create new InvoiceItem and Tax, associating both with the Invoice. 4. Prepare and send update request payload. 5. Invoke the updateInvoice method. 6. Assert updated Invoice and InvoiceItem present in the database with new data.
Test Data	- Existing Invoice created beforehand. - New Invoice data (number, sub_total, total, tax, discount, notes, template_name, customer_id). - One new InvoiceItem and Tax linked to the Invoice.
Expected Result	The Invoice and associated InvoiceItem and Tax are updated in the database, with all new data correctly reflected.
Actual Result	The Invoice and associated InvoiceItem and Tax are updated in the database, with all new data correctly reflected.
Status	Pass
Severity	High

Test ID	TC-IT-008
Title	Verify Creation of Invoice Items for an Existing Invoice
Objective	Confirm integrated workflow for adding multiple InvoiceItems to an existing Invoice and persisting them to the database.
Preconditions	- Application and relevant endpoints operational. - Database seeded with Invoice entity.
Steps	1. Create an Invoice using the factory. 2. Generate raw InvoiceItem linked to the Invoice. 3. Prepare item data and send via InvoicesRequest to the createItems method. 4. Assert the InvoiceItem is present in the database with all correct attributes.
Test Data	- Existing Invoice created using factory. - One raw InvoiceItem prepared for association.
Expected Result	The InvoiceItem is successfully linked and persisted to the existing Invoice in the database with all specified properties.
Actual Result	The InvoiceItem is successfully linked and persisted to the existing Invoice in the database with all specified properties.
Status	Pass

Severity	High

Test ID	TC-IT-009
Title	Verify Creation of Taxes for an Existing Invoice
Objective	Confirm integrated addition of Tax entities to an existing Invoice, ensuring correct database persistence and data setup.
Preconditions	- Application running. - Database seeded and contains Invoice entity.
Steps	1. Create an Invoice via factory. 2. Generate raw Tax data with invoice_id set. 3. Prepare tax data and send via Request to the createTaxes method. 4. Assert the Tax is present in the database with correct attributes.
Test Data	- Existing Invoice created via factory. - One raw Tax prepared for association.
Expected Result	The Tax entity is successfully linked and persisted to the existing Invoice in the database with provided properties.
Actual Result	The Tax entity is successfully linked and persisted to the existing Invoice in the database with provided properties.
Status	Pass
Severity	Medium

File: ItemTest.txt

Test ID	TC-IT-001
Title	Validate Item-to-Unit Relationship Mapping
Objective	Ensure that newly created items are correctly linked to their units, verifying the Item-Unit integration at the database level.
Preconditions	- Application environment is running. - Database has been seeded with base and demo data. - Factories for Item and Unit are available and correctly configured.
Steps	Step 1: Use factory to create an Item with an associated Unit. Step 2: Assert that the relationship between item and unit exists in the database.
Test Data	- Input: Creation of an Item instance via factory with an associated Unit. - Expected: The Item record has a valid unit relationship; unit() returns an existing Unit model.
Expected Result	The created Item is successfully linked to a Unit; item->unit() relationship returns an existing Unit model.
Actual Result	The created Item is successfully linked to a Unit; item->unit() relationship returns an existing Unit model.
Status	Pass
Severity	High

Test ID	TC-IT-002
Title	Verify Item-Tax Association Integrity for Multiple Taxes
Objective	Validate that items can be associated with multiple taxes and confirm the correct persistence and retrieval of these relationships.
Preconditions	- Application environment is running. - Database seeded with required base and demo data. - Factories for Item and Tax are available.
Steps	Step 1: Create an Item using factory and associate 5 Taxes. Step 2: Assert count of taxes attached to Item is 5. Step 3: Verify that the 'taxes' relationship exists for the Item.
Test Data	- Input: Creation of an Item instance via factory, associating it with 5 Taxes. - Expected: The Item has exactly 5 associated Tax models, and the 'taxes' relationship is correctly persisted.
Expected Result	Item record is associated with exactly 5 taxes, which are correctly persisted and retrievable via the 'taxes' relationship.
Actual Result	Item record is associated with exactly 5 taxes, which are correctly persisted and retrievable via the 'taxes' relationship.
Status	Pass
Severity	Medium

Test ID	TC-IT-003
Title	Confirm Item-to-Multiple Invoice Items Relationship
Objective	Ensure that items can be linked to multiple invoice items and these relationships are accurately reflected in the database.
Preconditions	- Application environment running. - Database seeded with base and demo data. - Factories for Item, InvoiceItem, and Invoice models available.
Steps	Step 1: Create an Item via factory and associate 5 InvoiceItems, each referencing a new Invoice. Step 2: Assert that the Item has 5 InvoiceItems. Step 3: Verify existence of invoiceItems() relationship for Item.

Test Data	- Input: Creation of an Item linked to 5 InvoiceItems, each connected to a new Invoice. - Expected: The Item has exactly 5 InvoiceItems associated with it and invoiceItems() relationship returns records.
Expected Result	Item is associated with 5 InvoiceItems, each correctly linked and retrievable; invoiceItems() relationship is present and populated.
Actual Result	Item is associated with 5 InvoiceItems, each correctly linked and retrievable; invoiceItems() relationship is present and populated.
Status	Pass
Severity	High

Test ID	TC-IT-004
Title	Validate Item-to-Multiple Estimate Items Association
Objective	Verify that an item can be associated with multiple estimate items, each linked to a distinct estimate, ensuring proper relational integrity.
Preconditions	- Application environment is running. - Database seeded with initial and demo data. - Factories for Item, EstimateItem, and Estimate models available.
Steps	Step 1: Create an Item using factory, associating it with 5 EstimateItems and 5 Estimates. Step 2: Assert that the number of EstimateItems for the Item is 5. Step 3: Confirm existence of estimateItems() relationship for created Item.
Test Data	- Input: Creation of an Item associated with 5 EstimateItems, each referencing a new Estimate. - Expected: The Item has 5 EstimateItems, and the estimateItems() relationship is working.
Expected Result	Item is linked to 5 EstimateItems, each properly associated and retrievable via the estimateItems() relationship.
Actual Result	Item is linked to 5 EstimateItems, each properly associated and retrievable via the estimateItems() relationship.
Status	Pass
Severity	Medium

File: PaymentMethodTest.txt

Test ID	TC-PMT-001
Title	Verify Payment Method Can Have Multiple Payments Associated
Objective	To validate that the Payment Method entity correctly establishes and maintains a one-to-many relationship with Payments, ensuring that multiple payments can be linked to a single payment method and that this relation is correctly retrieved.
Preconditions	- Application environment is running and all dependencies are installed. - Database is accessible and seeded with the required data using 'DatabaseSeeder' and 'DemoSeeder'. - Payment methods and payments model factories are available and properly configured.
Steps	Step 1: Create a Payment Method with 5 Payments using model factory. Step 2: Retrieve associated payments and verify that there are exactly 5. Step 3: Check that the payments relationship exists and is persisted in the database.
Test Data	- Inputs: Creation of a Payment Method instance with 5 associated payments via factory (e.g., PaymentMethod::factory()->hasPayments(5)->create()). - Expected values: Payment method should have exactly 5 payments linked; the relationship should exist in the database.
Expected Result	A Payment Method is created with exactly 5 associated payments; querying the payments relationship returns 5 records, and the relationship exists in the database.
Actual Result	A Payment Method is created with exactly 5 associated payments; querying the payments relationship returns 5 records, and the relationship exists in the database.
Status	Pass
Severity	High

Test ID	TC-PMT-002
Title	Verify Payment Method Belongs To Company
Objective	To ensure that each Payment Method is correctly linked to a Company via a belongs-to relationship, validating that every Payment Method references a valid existing Company in the system.
Preconditions	- Application environment is running and fully configured. - Database is accessible and seeded with 'DatabaseSeeder' and 'DemoSeeder'. - Payment Method and Company models and factories are available.
Steps	Step 1: Create a Payment Method using the factory. Step 2: Retrieve the associated company relationship instance. Step 3: Assert that the company relationship exists in the database.
Test Data	- Inputs: Creation of a Payment Method instance using factory (e.g., PaymentMethod::factory()->create()). - Expected values: The payment method should have a company relationship that exists and is retrievable.
Expected Result	A Payment Method is created and is associated to a Company; the company relationship can be queried and is present in the database.
Actual Result	A Payment Method is created and is associated to a Company; the company relationship can be queried and is present in the database.
Status	Pass
Severity	High

File: PaymentTest.txt

Test ID	TC-PT-001
Title	Validate Payment Association with Invoice
Objective	Verify that a Payment entity correctly establishes a relationship with its corresponding Invoice, ensuring referential consistency between payment and invoice records.
Preconditions	Application running with database connectivity. Database is seeded with both base and demo data via DatabaseSeeder and DemoSeeder. Payment, Invoice, and necessary related models are available.
Steps	Step 1: Trigger Payment factory creation using forInvoice() to associate with an invoice. Step 2: Verify that the invoice relationship exists for the Payment by asserting existence in the database.
Test Data	Payment record generated via factory using forInvoice() to target invoice relationship. Expected: Newly created Payment must reference a valid Invoice entity.
Expected Result	The Payment record has a linked Invoice entity. The payment->invoice() relation exists, confirming payment-to-invoice association is correctly established.
Actual Result	The Payment record has a linked Invoice entity. The payment->invoice() relation exists, confirming payment-to-invoice association is correctly established.
Status	Pass
Severity	High

Test ID	TC-PT-002
Title	Validate Payment Association with Customer
Objective	Verify that a Payment entity correctly establishes a relationship with its corresponding Customer, ensuring that payments are properly attributed to customers in the system.
Preconditions	Application running with database access. Database seeded using DatabaseSeeder and DemoSeeder to provide necessary customer records. Relevant models loaded: Payment, Customer.
Steps	Step 1: Trigger Payment factory creation using forCustomer() to associate with a customer. Step 2: Confirm that the customer relationship exists for the Payment by checking for existence in the database.
Test Data	Payment record generated via factory using forCustomer() to target customer relationship. Expected: Newly created Payment must reference a valid Customer entity.
Expected Result	The Payment record has a linked Customer entity. The payment->customer() relation successfully exists, validating proper assignment of payments to customers.
Actual Result	The Payment record has a linked Customer entity. The payment->customer() relation successfully exists, validating proper assignment of payments to customers.
Status	Pass
Severity	High

Test ID	TC-PT-003
Title	Validate Payment Association with Payment Method

Objective	Ensure that a Payment entity is correctly associated with a Payment Method, allowing tracking of the payment method used during the transaction.
Preconditions	Application is running and database is accessible. Database has been seeded with test and demo data via DatabaseSeeder and DemoSeeder, including Payment Methods. Related models available: Payment, PaymentMethod.
Steps	Step 1: Create a Payment using factory with forPaymentMethod(), thereby associating it with a payment method. Step 2: Verify that the paymentMethod relationship exists for the Payment by asserting record existence in the database.
Test Data	Payment record generated via factory using forPaymentMethod() to establish payment method relationship. Expected: Newly created Payment references a valid Payment Method.
Expected Result	The Payment record is associated with a valid Payment Method entity. The payment->paymentMethod() relation exists, confirming payment tracking by method.
Actual Result	The Payment record is associated with a valid Payment Method entity. The payment->paymentMethod() relation exists, confirming payment tracking by method.
Status	Pass
Severity	Medium

File: RecurringInvoiceTest.txt

Test ID	TC-RIT-001
Title	Verify Recurring Invoice- Invoice Relationship
Objective	Validate that a Recurring Invoice correctly establishes and retrieves multiple associated Invoice records, confirming many-to-one relationship at the database layer.
Preconditions	- Application and database instance running. - All relevant migrations applied and database schema up to date. - Database seeded with base and demo data (DatabaseSeeder, DemoSeeder).
Steps	Step 1: Create a RecurringInvoice using the factory with 5 associated invoices. Step 2: Assert that the RecurringInvoice's invoices collection contains exactly 5 records. Step 3: Assert that the invoices relationship on the RecurringInvoice exists (not empty).
Test Data	- RecurringInvoice factory creates a RecurringInvoice with 5 associated Invoices. - Expected: RecurringInvoice has exactly 5 Invoices and invoices relationship exists.
Expected Result	RecurringInvoice object has exactly 5 associated Invoice records; invoices() relationship returns a collection with five items and exists() returns true.
Actual Result	RecurringInvoice object has exactly 5 associated Invoice records; invoices() relationship returns a collection with five items and exists() returns true.
Status	Pass
Severity	High

Test ID	TC-RIT-002
Title	Verify Recurring Invoice- Invoice Item Relationship
Objective	Validate that a Recurring Invoice accurately creates and retrieves multiple associated Invoice Item records, confirming proper linkage and data integrity.
Preconditions	- Application and database instance running. - All migrations and seeders applied for consistent structure.
Steps	Step 1: Create a RecurringInvoice using the factory with 5 associated invoice items. Step 2: Assert that the RecurringInvoice's items collection contains exactly 5 records. Step 3: Assert that the items relationship on the RecurringInvoice exists (not empty).
Test Data	- RecurringInvoice factory creates a RecurringInvoice with 5 associated items. - Expected: RecurringInvoice has exactly 5 Items and items relationship exists.
Expected Result	RecurringInvoice object retrieves a collection of 5 associated Invoice Items, and items() relationship exists, confirming correct association.
Actual Result	RecurringInvoice object retrieves a collection of 5 associated Invoice Items, and items() relationship exists, confirming correct association.
Status	Pass
Severity	Medium

Test ID	TC-RIT-003
Title	Verify Recurring Invoice- Tax Relationship
Objective	Ensure that a Recurring Invoice can successfully associate and retrieve multiple tax records, verifying many-to-one integration and correct persistence.

Preconditions	- Application running with clean database. - Database seeded with base and demo data using required seeders.
Steps	Step 1: Create a RecurringInvoice with 5 related tax entities via factory. Step 2: Assert the RecurringInvoice's taxes collection contains exactly 5 records. Step 3: Assert that the taxes relationship collection exists and returns at least one item.
Test Data	- RecurringInvoice factory creates a RecurringInvoice with 5 associated taxes. - Expected: RecurringInvoice has exactly 5 Taxes and taxes relationship exists.
Expected Result	RecurringInvoice object successfully retrieves 5 associated Tax records, and taxes() relationship exists, demonstrating valid integration.
Actual Result	RecurringInvoice object successfully retrieves 5 associated Tax records, and taxes() relationship exists, demonstrating valid integration.
Status	Pass
Severity	Medium

Test ID	TC-RIT-004
Title	Verify Recurring Invoice Association with Customer
Objective	Confirm that a Recurring Invoice is properly linked to a Customer record, validating one-to-one relationship integrity between invoice and customer entities.
Preconditions	- Application and database are running and accessible. - Database seeded using DatabaseSeeder and DemoSeeder.
Steps	Step 1: Use factory to create RecurringInvoice with an attached customer. Step 2: Assert the customer() relationship on RecurringInvoice exists and is correctly populated.
Test Data	- RecurringInvoice factory creates a RecurringInvoice associated with a customer. - Expected: RecurringInvoice has an existing customer relationship.
Expected Result	RecurringInvoice object returns a valid customer relationship; customer() exists() returns true, confirming linkage to a customer entity.
Actual Result	RecurringInvoice object returns a valid customer relationship; customer() exists() returns true, confirming linkage to a customer entity.
Status	Pass
Severity	High

File: SettingTest.txt

Test ID	TC-ST-001
Title	Setting Creation and Retrieval Integration
Objective	Validate that a setting can be created via the Setting model and immediately retrieved, ensuring persistence and correct value mapping between data layer and application logic.
Preconditions	- Application is running and Laravel environment is correctly set up. - Database is seeded with initial data using DatabaseSeeder and DemoSeeder. - Setting model and necessary tables are present and accessible.
Steps	Step 1: Seed the database with necessary data (DatabaseSeeder and DemoSeeder). Step 2: Generate a random key and value using Faker. Step 3: Invoke Setting::setSetting($key, $value) to store the setting. Step 4: Retrieve the stored setting using Setting::getSetting($key). Step 5: Assert that the retrieved value matches the original value set.
Test Data	- Input: Randomly generated key (faker()->name) and value (faker()->word) for the setting. - Expected Value: The same value used for setting, to be returned upon retrieval.
Expected Result	The value retrieved through Setting::getSetting($key) exactly matches the value previously set via Setting::setSetting($key); demonstrates successful integration between the model and persistent storage.
Actual Result	The value retrieved through Setting::getSetting($key) exactly matches the value previously set via Setting::setSetting($key); demonstrates successful integration between the model and persistent storage.
Status	Pass
Severity	Medium

File: TaxTest.txt

Test ID	TC-TT-001
Title	Verify Tax is linked to Tax Type entity
Objective	Validate that each Tax created in the system has an associated Tax Type, ensuring relational linking in the database.
Preconditions	- Application running in test environment - Database seeded with required base and demo data (DatabaseSeeder, DemoSeeder) - Factories for Tax and TaxType are configured
Steps	Step 1: Create a Tax entity using the factory Step 2: Retrieve the taxonomy (taxType) relation for the created Tax Step 3: Verify Tax->taxType relation returns a valid (existent) object in the database
Test Data	- Input: Creation of Tax model via factory - Expected Value: Tax->taxType relation should exist (record in tax_types table linked with tax)
Expected Result	Tax entity is correctly related to a TaxType. Querying $tax->taxType returns an existing associated record, validating the system's referential integrity.
Actual Result	Tax entity is correctly related to a TaxType. Querying $tax->taxType returns an existing associated record, validating the system's referential integrity.
Status	Pass
Severity	High

Test ID	TC-TT-002
Title	Verify Tax association with an Invoice record
Objective	Ensure that a Tax can be linked to an Invoice, confirming the relationship is established in the database for taxation purposes.
Preconditions	- Application running - Database seeded with all necessary entities - Invoice and Tax factories available
Steps	Step 1: Use Tax factory with forInvoice() method to create a new Tax tied to an Invoice Step 2: Retrieve Tax->invoice() relationship Step 3: Confirm the relationship returns an existing Invoice and not null
Test Data	- Input: Tax factory creation using 'forInvoice', providing linkage to a new Invoice record - Expected Value: Tax->invoice() relation exists, returns a valid Invoice
Expected Result	The Tax's invoice relationship exists and returns a valid Invoice instance, ensuring association for invoice taxation.
Actual Result	The Tax's invoice relationship exists and returns a valid Invoice instance, ensuring association for invoice taxation.
Status	Pass
Severity	High

Test ID	TC-TT-003
Title	Verify Tax association with Recurring Invoice entity
Objective	Confirm that Tax objects can be tied to Recurring Invoices, supporting business requirements for ongoing billing.
Preconditions	- Application running - Database pre-seeded - Factories for Tax and Recurring Invoice enabled

Steps	Step 1: Create Tax using factory with forRecurringInvoice() linkage Step 2: Access Tax->recurringInvoice() relation Step 3: Assert that Tax is associated with an existing Recurring Invoice
Test Data	- Input: Tax factory setup using forRecurringInvoice - Expected Value: Tax->recurringInvoice() returns a valid Recurring Invoice
Expected Result	Tax is associated with a valid Recurring Invoice object, confirming system supports recurring billing taxation.
Actual Result	Tax is associated with a valid Recurring Invoice object, confirming system supports recurring billing taxation.
Status	Pass
Severity	Medium

Test ID	TC-TT-004
Title	Validate Tax linkage to an Estimate record
Objective	Check that Taxes can be associated with Estimates to handle preliminary taxation calculations.
Preconditions	- Application running - Database seeded with Estimates and supporting entities - Tax and Estimate factories available
Steps	Step 1: Create a Tax using factory and link it to an Estimate via forEstimate() Step 2: Access Tax->estimate() relationship Step 3: Assert relationship returns an associated Estimate record
Test Data	- Input: Tax factory with forEstimate() linkage - Expected Value: Tax->estimate() returns valid Estimate
Expected Result	Tax is correctly linked to an Estimate record; $tax->estimate() retrieves an existing Estimate.
Actual Result	Tax is correctly linked to an Estimate record; $tax->estimate() retrieves an existing Estimate.
Status	Pass
Severity	Medium

Test ID	TC-TT-005
Title	Validate Tax is associated with an Invoice Item entity
Objective	Ensure Taxes are assignable at the invoice item level, supporting fine-grained tax calculations.
Preconditions	- Application running - Database seeded with Invoice, InvoiceItem entities - Factories for Tax, Invoice, and InvoiceItem set up
Steps	Step 1: Use Tax factory to create and bind a Tax to an InvoiceItem (with its own Invoice) Step 2: Access Tax->invoiceItem() relationship Step 3: Validate that relationship returns a valid InvoiceItem instance
Test Data	- Input: Tax factory linked to an InvoiceItem, with invoice_id state set via factory - Expected Value: Tax->invoiceItem() returns a valid InvoiceItem object
Expected Result	Tax entity successfully links to an existing InvoiceItem, supporting item-level tax application.
Actual Result	Tax entity successfully links to an existing InvoiceItem, supporting item-level tax application.
Status	Pass
Severity	High

Test ID	TC-TT-006
Title	Validate Tax is associated with an Estimate Item entity
Objective	Verify Taxes are applicable at the estimate item level for detailed taxation before conversion to invoice.
Preconditions	- Application running - Database seeded with Estimate, EstimateItem entities - Factories for Tax, Estimate, and EstimateItem set up
Steps	Step 1: Create Tax using factory, linked to an EstimateItem (and its Estimate) Step 2: Access Tax->estimateItem() relationship Step 3: Assert it returns an existing EstimateItem
Test Data	- Input: Tax factory bound to an EstimateItem (with estimate_id set via factory) - Expected Value: Tax->estimateItem() relation yields a valid EstimateItem
Expected Result	Tax entity is linked with an EstimateItem, supporting detailed pre-invoice tax calculations.
Actual Result	Tax entity is linked with an EstimateItem, supporting detailed pre-invoice tax calculations.
Status	Pass
Severity	Medium

Test ID	TC-TT-007
Title	Verify Tax association with generic Item entity
Objective	Ensure that various items in the system can have associated Tax records for proper taxation.
Preconditions	- Application running - Database seeded with Item entities - Tax and Item factories configured
Steps	Step 1: Create Tax with factory using forItem() association Step 2: Access Tax->item() relationship Step 3: Confirm that Tax is correctly associated with an existing Item entity
Test Data	- Input: Tax factory created with forItem() association - Expected Value: Tax->item() returns valid Item reference
Expected Result	Tax record is associated with an Item entity, supporting flexible taxation across products or services.
Actual Result	Tax record is associated with an Item entity, supporting flexible taxation across products or services.
Status	Pass
Severity	Medium

File: TaxTypeTest.txt

Test ID	TC-TTT-001
Title	Verify TaxType association with multiple Taxes
Objective	Validate that the TaxType entity correctly establishes and persists a one-to-many relationship with associated Taxes in the database.
Preconditions	- Application is running and the necessary environment variables are set. - Database is seeded with required data via DatabaseSeeder and DemoSeeder. - Factory definitions for TaxType and Tax are configured and available.
Steps	Step 1: Use the TaxType factory to create a new TaxType with 4 related Tax entities. Step 2: Retrieve the taxes relation and assert that its count equals 4. Step 3: Assert that the taxes relationship exists for the created TaxType in the database.
Test Data	- Input: Creation of a TaxType entity using the factory, configured to have 4 related Tax entities. - Expected values: TaxType should have exactly 4 associated Taxes; the relationship should exist in the database.
Expected Result	The created TaxType has exactly 4 associated Taxes; the hasMany relationship is correctly implemented and persisted as verified by both the in-memory and database relationship checks.
Actual Result	The created TaxType has exactly 4 associated Taxes; the hasMany relationship is correctly implemented and persisted as verified by both the in-memory and database relationship checks.
Status	Pass
Severity	Medium

File: UnitTest.txt

Test ID	TC-UT-001
Title	Verify Unit-Item Relationship (Unit Has Many Items)
Objective	Validate that a Unit record can have multiple associated Item records and the integration between Unit factory creation and Item association functions as expected.
Preconditions	- Application environment running and configured for integration testing. - The database has been seeded with both DatabaseSeeder and DemoSeeder using Artisan. - A 'super admin' user exists, is authenticated via Sanctum, and request headers are set for the appropriate company context.
Steps	Step 1: Create a Unit using the factory with 5 associated Item records. Step 2: Query the Unit for its related Items. Step 3: Verify that the Unit has associated Item records (relationship exists).
Test Data	- Factory-generated Unit with 5 associated Items. - Expected value: Unit has at least one associated Item after creation.
Expected Result	The created Unit has one or more associated Item records; the relationship exists in the database, confirming that the Unit-Item association is properly integrated.
Actual Result	The created Unit has one or more associated Item records; the relationship exists in the database.
Status	Pass
Severity	High

Test ID	TC-UT-002
Title	Verify Unit-Company Relationship (Unit Belongs to Company)
Objective	Validate that a Unit record is properly associated with a Company, ensuring the Unit-Company relationship is created and persisted correctly in integrated systems.
Preconditions	- Application environment running and ready for integration testing. - Database has been seeded with DatabaseSeeder and DemoSeeder via Artisan. - A 'super admin' user is present, authenticated through Sanctum, with request context set for their company.
Steps	Step 1: Generate a Unit using the factory within the seeded environment. Step 2: Query the Unit for its related Company. Step 3: Confirm that the Unit is associated with a Company (relationship exists).
Test Data	- Factory-generated Unit record (belongs to seeded company). - Expected value: Unit is associated with a Company record after creation.
Expected Result	The created Unit is correctly related to a Company; querying the relationship retrieves an existing Company record, confirming Unit-Company integration functions as intended.
Actual Result	The created Unit is correctly related to a Company; querying the relationship retrieves an existing Company record.
Status	Pass
Severity	High

File: UserTest.txt

Test ID	TC-UT-001
Title	Verify User-Currency Relationship Existence
Objective	Ensure that a newly created user is assigned a currency and the relationship exists at the database-model integration level.
Preconditions	- Application is running with the required PHP/Laravel environment. - Database is seeded with DatabaseSeeder and DemoSeeder to ensure required reference data exists. - User model and Currency model relationships are properly defined and migrated.
Steps	Step 1: Seed the database with essential data using DatabaseSeeder and DemoSeeder. Step 2: Use the User factory to create a new user record. Step 3: Retrieve the currency relationship from the newly created user using $user->currency(). Step 4: Check if the currency relationship exists with $user->currency()->exists().
Test Data	- Input: New User instance created via User::factory()->create() - Validation: Assert that $user->currency()->exists() returns true, confirming that the user has an associated currency.
Expected Result	After creation, the User instance will have a valid currency relationship; $user->currency()->exists() returns true, confirming user-currency integration at the model/database layer.
Actual Result	After creation, the User instance has a valid currency relationship; $user->currency()->exists() returns true, confirming user-currency integration at the model/database layer.
Status	Pass
Severity	High

Test ID	TC-UT-002
Title	Verify User-Company Multiple Relationship Integrity
Objective	Ensure that a user can belong to multiple companies and that the companies relationship returns a valid Eloquent Collection, verifying many-to-many integration.
Preconditions	- Application is operating in the correct environment (PHP/Laravel). - Database has been seeded with DatabaseSeeder and DemoSeeder for required reference records. - User and Company models support many-to-many relationships; pivot tables are migrated.
Steps	Step 1: Seed the database with DatabaseSeeder and DemoSeeder for prerequisite data. Step 2: Create a new User record with 5 related Company instances using User factory. Step 3: Access the companies relationship on the user ($user->companies). Step 4: Validate that the companies relationship returns an Eloquent Collection object representing multiple linked companies.
Test Data	- Input: New User with 5 associated companies created via User::factory()->hasCompanies(5)->create() - Validation: Assert that $user->companies returns an instance of Illuminate\Database\Eloquent\Collection, confirming that multiple companies are correctly linked.
Expected Result	User instance is associated with 5 companies, and $user->companies returns a valid Eloquent Collection, confirming many-to-many relationship integration.
Actual Result	User instance is associated with 5 companies, and $user->companies returns a valid Eloquent Collection, confirming many-to-many relationship integration.

Status	Pass
Severity	High