a

Proposed Features & Controller Implementation Guide

This document is split into two sections. Section 1 lists every feature proposed in the HRMS system documentation (Chapters 1, 2, 3 and 4), with a clear description of what each feature does. Section 2 describes how each feature is implemented in Laravel controllers using your specific tool stack, with learning notes explaining the key concepts you need to understand at each step.

osed Features and Descriptions

Sourced from Chapter 1 Scope, Chapter 3 Modules, and Ch

1 Secure User Authentication

Chapter 1 Scope · Chapter 4.8.4

The system requires every user to log in with a username and password before accessing any feature. Authentication is token-based using Laravel Sanctum, meaning a secure API token is issued on login and must be sent with every subsequent request. Sessions time out automatically after inactivity. Passwords are never stored in plain text — they are hashed using bcrypt. Only active accounts can authenticate; deactivated accounts are rejected.

2 Role-Based Access Control (RBAC)

Chapter 1 Scope · Chapter 4.8.4

Every user is assigned one of four roles: HR Administrator, Branch Manager, HR Officer, or Auditor. Each role determines which endpoints the user can call and which records they can see or modify. For example, only HR Administrators can deactivate employee accounts or access payroll data. Branch Managers see only employees within their branch. Auditors have read-only access to records and the audit log. Role permissions are enforced at the API layer, meaning they cannot be bypassed by manipulating the frontend.

3 Employee Profile Management

Chapter 1 Scope · Chapter 4.8.1

HR staff can create, view, update, and deactivate employee profiles through the system. Each profile captures personal details (name, date of birth, national ID, contact information), employment details (job title, hire date, employment status, department, branch, supervisor), and references to uploaded documents. Profiles are never permanently deleted — deactivated profiles remain in the database for historical and audit purposes. The system enforces uniqueness on National ID and Email to prevent duplicate records.

4 Digital Document Upload and Storage

Chapter 1 Scope · Chapter 4.8.2

HR personnel can upload employee documents such as national ID copies, academic certificates, good conduct certificates, employment contracts, and security licences. Files are stored securely in Backblaze B2 cloud storage rather than on the local server. Each document is categorised by document type, linked to the employee record, stamped with an upload date, attributed to the uploading HR officer, and optionally given an expiry date for licence and certification tracking. Documents can be downloaded by authorised users through secure, time-limited signed URLs that expire after a short period.

5 Search and Filtering

Chapter 1 Scope · Chapter 4.8.3

Authorised users can search employee records by name, national ID number, employee number, department, branch, employment status, or job title. Multiple filters can be combined in a single query. Results must be returned in under five seconds regardless of database size. Filtered results can be exported to PDF or CSV for reporting. The search is case-insensitive and supports partial matches so that typing "kam" finds "John Kamau".

6 Leave Request and Approval

Chapter 3 ERD Modules

Employees can submit leave requests specifying the leave type (annual, sick, maternity, emergency), start and end dates, and a reason. The system validates that the end date is not before the start date and checks the employee's leave balance before accepting the request. A Branch Manager or HR Administrator can then approve or reject the request. Leave balances are automatically updated when a request is approved. The system tracks who approved each request and when.

7 Attendance Tracking

Chapter 3 ERD Modules

The system records daily attendance for each employee, capturing the date, time-in, time-out, and attendance status (Present, Absent, Late, Half-Day). Attendance records are linked to the employee and can be filtered by date range or employee. This data feeds into payroll processing and performance evaluation. Only HR Officers and above can create or modify attendance records.

8 Payroll Processing

Chapter 3 ERD Modules

The system calculates employee net salary for each pay period by combining basic salary with all assigned allowances (housing, transport, medical) and subtracting all assigned deductions (NHIF, NSSF, PAYE). Each payroll run generates a record linked to the employee, the pay period, the applied allowances, and the applied deductions. The calculation is handled in a dedicated service class to keep the controller thin. Only HR Administrators can generate or view payroll records.

9 Performance Evaluation

Chapter 3 ERD Modules

HR staff and supervisors can create performance evaluations for employees, recording the evaluation period, a score between 1 and 100, and written comments. Each evaluation captures who conducted it (the evaluator) and who it is about (the employee). Employees can view their own evaluations. Supervisors can view evaluations for their direct reports. Only HR Administrators can view all evaluations across the organisation.

10 Training Management

Chapter 3 ERD Modules

The system maintains a catalogue of training programmes with name, type, and schedule. Employees can be enrolled in training programmes through a many-to-many relationship tracked in the EmployeeTraining junction table. Completion status is recorded per employee per training. HR staff can report on which employees have completed mandatory training and which have outstanding enrolments.

11 Recruitment and Applicant Management

Chapter 3 ERD Modules

HR Administrators can create recruitment postings linked to a specific department with a job title, vacancy status, and posted date. Applicants submit their details against a specific recruitment posting, including personal information and file paths for their uploaded CV, letter of application, and good conduct certificate. Shortlisted applicants can be converted into employee records when hired.

12 Guard Deployment History

Chapter 2.6 & 2.7 — Security Sector Specific

This feature is unique to GardaWorld as a security company and was identified in Chapter 2 as absent from all commercial HRMS platforms reviewed. The system tracks every deployment of a security guard to a site or branch, recording the location, start date, end date, reason for deployment, and who authorised it. This enables branch managers to know exactly where each guard has been deployed historically and which guards are currently active at each site.

13 Audit Logging

Chapter 1 Scope · Chapter 4.8.5 · Chapter 4.9

Every significant action in the system — creating, updating, viewing, or deleting any record — is automatically recorded in a tamper-evident audit log. Each log entry captures the action type, the table and record affected, the data before and after the change, the user who performed the action, their IP address, and a timestamp. Audit logs cannot be edited or deleted through the system interface. This satisfies both the Chapter 4 non-functional requirements and the Kenya Data Protection Act 2019 accountability obligations.

14 System Notifications and Alerts

Chapter 1 Scope · Chapter 4.8.5

The system automatically generates notifications in two scenarios: event-driven alerts (a leave request is approved, a document is uploaded) and scheduled expiry alerts (a security licence or certificate is approaching its expiry date). Notifications are delivered in two channels: in-system (stored in the Notifications table and pushed to the user interface in real time via Laravel Reverb WebSockets) and via email (sent through Mailgun). Notifications are marked as read when the user views them.

15 User Account Management

Chapter 4.8.4

HR Administrators can create user accounts linked to employee records, assign roles, activate or deactivate accounts, and reset passwords. A user account is separate from an employee profile — an employee exists in the system before they are given login access. Deactivating an account prevents login but preserves all records associated with that user. Account activity is tracked through the LastLogin timestamp updated on every successful login.

Every implementation below follows the MVC pattern enforced by Laravel. The controller receives the request, delegates to a model or service, and returns a JSON response. Business logic that is too complex for a controller is placed in a Service class inside app/Services/. Routes are registered in routes/api.php and protected by the auth:sanctum middleware.

1 Secure User Authentication

AuthController.php · Laravel Sanctum

How it works

On login, the controller receives a username and password, finds the matching UserAccount record, verifies the password against the stored bcrypt hash using Hash::check(), then calls createToken() which writes a hashed token into the personal_access_tokens table and returns the plain-text token once to the client. The client stores this token and sends it as a Bearer token in the Authorization header of every subsequent request. Laravel Sanctum's auth:sanctum middleware intercepts each request, finds the token in the database, loads the user, and makes it available via $request->user().

app/Http/Controllers/AuthController.php — login method

```
public function login(Request $request): JsonResponse
{
    $request->validate([
        'Username' => 'required|string',
        'password' => 'required|string',
    ]);

    $user = UserAccount::where('Username', $request->Username)
                       ->where('AccountStatus', 'active')
                       ->first();

    if (!$user || !Hash::check($request->password, $user->PasswordHash)) {
        return response()->json(['message' => 'Invalid credentials'], 401);
    }

    $user->LastLogin = now();
    $user->save();

    $token = $user->createToken('hrms-token')->plainTextToken;

    return response()->json([
        'token' => $token,
        'user'  => $user->load('role', 'employee'),
    ]);
}
```

LEARNING NOTE — Hash::check() compares a plain-text password against a bcrypt hash without you ever seeing the original password. Laravel never stores plain passwords anywhere. createToken() returns a NewAccessToken object — you access the string you send to the client via ->plainTextToken. After that call, the plain text is gone; only the hash is stored. The ->load() call on the user is called eager loading — it fetches the role and employee relations in the same response without a second API call from the client.

2 Role-Based Access Control

RoleMiddleware.php · routes/api.php

How it works

A custom middleware class checks the authenticated user's role before the request reaches the controller. The middleware is registered in bootstrap/app.php with an alias, then applied to specific route groups in api.php. The middleware reads the user's RoleID via $request->user()->role->RoleName and rejects the request with a 403 Forbidden if the role does not match.

app/Http/Middleware/RoleMiddleware.php

```
public function handle(Request $request, Closure $next, string ...$roles): Response
{
    $user = $request->user();

    if (!$user || !$user->role) {
        return response()->json(['message' => 'Unauthorised'], 403);
    }

    if (!in_array($user->role->RoleName, $roles)) {
        return response()->json(['message' => 'Forbidden: insufficient role'], 403);
    }

    return $next($request);
}
```

routes/api.php — applying role middleware to a route group

```
Route::middleware(['auth:sanctum', 'role:HR Administrator'])->group(function () {
    Route::apiResource('payroll', PayrollController::class);
    Route::delete('employees/{id}', [EmployeeController::class, 'destroy']);
});

Route::middleware(['auth:sanctum', 'role:HR Administrator,Branch Manager'])->group(function () {
    Route::patch('leave/{id}/approve', [LeaveController::class, 'approve']);
});
```

bootstrap/app.php — registering the middleware alias

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
        'role' => \App\Http\Middleware\RoleMiddleware::class,
    ]);
})
```

LEARNING NOTE — Middleware runs before the controller. Think of it as a security gate on the route. The string ...$roles parameter uses PHP variadic syntax, meaning you can pass one or more roles in the route definition separated by commas: role:HR Administrator,Branch Manager. The in_array() check tests whether the user's role is in that list. If the check fails, the request never reaches the controller — Laravel returns the 403 immediately.

3 Employee Profile Management

EmployeeController.php · Employee model

How it works

The EmployeeController is a resource controller generated by artisan. It provides index (list), store (create), show (single record), update, and destroy (deactivate) methods. Validation is handled by a dedicated Form Request class. The destroy method does not delete the record — it sets EmploymentStatus to Inactive, preserving the record for audit purposes.

Terminal — generate the controller and form request

```
sail artisan make:controller EmployeeController --api
sail artisan make:request StoreEmployeeRequest
sail artisan make:request UpdateEmployeeRequest
```

app/Http/Requests/StoreEmployeeRequest.php — validation rules

```
public function rules(): array
{
    return [
        'FullName'         => 'required|string|max:200',
        'NationalID'       => 'required|string|unique:Employee,NationalID',
        'Email'            => 'nullable|email|unique:Employee,Email',
        'HireDate'         => 'required|date',
        'DepartmentID'     => 'required|exists:Department,DepartmentID',
        'BranchID'         => 'required|exists:Branch,BranchID',
        'EmploymentStatus' => 'required|in:Active,Inactive,Suspended',
    ];
}
```

app/Http/Controllers/EmployeeController.php

```
public function store(StoreEmployeeRequest $request): JsonResponse
{
    $employee = Employee::create($request->validated());
    return response()->json($employee->load('department', 'branch'), 201);
}

public function show(int $id): JsonResponse
{
    $employee = Employee::with([
        'department', 'branch', 'supervisor',
        'documents.documentType', 'leaveBalances',
    ])->findOrFail($id);

    return response()->json($employee);
}

public function destroy(int $id): JsonResponse
{
    $employee = Employee::findOrFail($id);
    $employee->update(['EmploymentStatus' => 'Inactive']);
    return response()->json(['message' => 'Employee deactivated']);
}
```

LEARNING NOTE — findOrFail() automatically returns a 404 JSON response if the record does not exist, so you do not need to write that check yourself. The unique:Employee,NationalID validation rule queries the Employee table and rejects the request if that NationalID already exists — preventing duplicate records at the API level before the data even reaches the database. Using $request->validated() instead of $request->all() means only fields that passed validation are passed to the model, protecting against mass assignment of unexpected fields.

4 Document Upload and Storage

`DocumentController.php · Backblaze B2`

How it works

Backblaze B2 exposes an S3-compatible API. Laravel's Storage facade communicates with it using the league/flysystem-aws-s3-v3 package, configured with B2's custom endpoint. When a file is uploaded, the controller uses Storage::disk('b2') to stream the file directly to B2 and stores only the returned file path in the AdditionalDocuments table. For downloads, a temporary signed URL is generated that expires after a short window, preventing public access to sensitive documents.

Terminal — install the S3 filesystem driver

```
sail composer require league/flysystem-aws-s3-v3
```

config/filesystems.php — add the b2 disk

```
"b2" => [
    "driver"                 => "s3",
    "key"                    => env("B2_KEY_ID"),
    "secret"                 => env("B2_APP_KEY"),
    "region"                 => env("B2_REGION", "us-west-002"),
    "bucket"                 => env("B2_BUCKET"),
    "endpoint"               => env("B2_ENDPOINT"),
    "use_path_style_endpoint" => true,
    "visibility"             => "private",
],
```

.env — Backblaze B2 credentials

```
B2_KEY_ID=your_key_id
B2_APP_KEY=your_application_key
B2_BUCKET=your_bucket_name
B2_REGION=us-west-002
B2_ENDPOINT=https://s3.us-west-002.backblazeb2.com
```

app/Http/Controllers/DocumentController.php

```
public function store(Request $request, int $employeeId): JsonResponse
{
    $request->validate([
        'file'           => 'required|file|mimes:pdf,jpg,jpeg,png|max:10240',
        'DocumentTypeID' => 'required|exists:DocumentType,DocumentTypeID',
        'Description'    => 'nullable|string',
        'ExpiryDate'     => 'nullable|date|after:today',
    ]);

    $employee = Employee::findOrFail($employeeId);

    $path = $request->file("file")->store(
        "employees/{$employeeId}/documents",
        "b2"
    );

    $document = AdditionalDocuments::create([
        "EmployeeID"     => $employeeId,
        "DocumentTypeID" => $request->DocumentTypeID,
        "Document"       => $path,
        "Description"    => $request->Description,
        "ExpiryDate"     => $request->ExpiryDate,
        "UploadDate"     => now(),
        "UploadedBy"     => $request->user()->EmployeeID,
    ]);

    return response()->json($document->load("documentType"), 201);
}

public function show(int $documentId): JsonResponse
{
    $doc = AdditionalDocuments::findOrFail($documentId);
    $url = Storage::disk("b2")->temporaryUrl($doc->Document, now()->addMinutes(5));
    return response()->json(["url" => $url]);
}
```

LEARNING NOTE — Storage::disk("b2") tells Laravel to use the b2 disk you configured in filesystems.php. The ->store() method handles the entire upload: it reads the file from the request, streams it to B2, and returns the stored path as a string. You never touch the file contents in PHP — Laravel handles the streaming. The temporaryUrl() method generates a pre-signed S3 URL with an expiry time. After 5 minutes, the URL stops working — the file in B2 is never publicly accessible. The mimes validation rule checks the actual file content type, not just the extension, preventing disguised file uploads.

5 Search and Filtering

EmployeeController.php — index method · Eloquent query scopes

How it works

The index method of EmployeeController reads optional query parameters from the request URL and builds a dynamic Eloquent query using conditional where clauses. Laravel's query builder chains methods, so each filter is only applied when the corresponding parameter is present. Results are paginated using Laravel's built-in paginate() method which automatically handles page and per_page query parameters.

app/Http/Controllers/EmployeeController.php — index method

```
public function index(Request $request): JsonResponse
{
    $query = Employee::with(["department", "branch"]);

    if ($request->filled("search")) {
        $term = $request->search;
        $query->where(function ($q) use ($term) {
            $q->where("FullName",  "like", "%{$term}%")
              ->orWhere("NationalID",    "like", "%{$term}%")
              ->orWhere("Email",         "like", "%{$term}%")
              ->orWhere("JobTitle",      "like", "%{$term}%");
        });
    }

    if ($request->filled("branch_id")) {
        $query->where("BranchID", $request->branch_id);
    }

    if ($request->filled("department_id")) {
        $query->where("DepartmentID", $request->department_id);
    }

    if ($request->filled("status")) {
        $query->where("EmploymentStatus", $request->status);
    }

    $employees = $query->paginate($request->get("per_page", 20));

    return response()->json($employees);
}
```

Example Postman GET request with filters

```
GET /api/employees?search=kamau&branch_id=1&status=Active&page=1&per_page=15
```

LEARNING NOTE — The filled() method checks that a parameter is present AND not empty, so ?search= (empty string) does not trigger the search clause. The nested where closure groups the OR conditions so they do not interfere with the AND filters — without the closure, the branch_id filter would be OR'd with the search term instead of AND'd. paginate() returns a LengthAwarePaginator object which serialises to JSON including total, current_page, last_page, and the data array — the frontend uses this to render pagination controls without any extra work.

6 Leave Request and Approval

LeaveController.php

How it works

The store method validates and creates a leave request with a pending status. The approve method is a separate endpoint that only HR Administrators and Branch Managers can reach (enforced by the role middleware). When

approved, the system increments UsedDays in LeaveBalance for that employee and leave type — the RemainingDays column updates automatically because it is a generated column in MySQL.

app/Http/Controllers/LeaveController.php

```
public function store(Request $request): JsonResponse
{
    $validated = $request->validate([
        "EmployeeID" => "required|exists:Employee,EmployeeID",
        "LeaveType"  => "required|string",
        "StartDate"  => "required|date",
        "EndDate"    => "required|date|after_or_equal:StartDate",
        "Reason"     => "nullable|string",
    ]);

    $balance = LeaveBalance::where("EmployeeID", $validated["EmployeeID"])
                           ->where("LeaveType",  $validated["LeaveType"])
                           ->first();

    $days = Carbon::parse($validated["StartDate"])
                  ->diffInWeekdays(Carbon::parse($validated["EndDate"])) + 1;

    if (!$balance || $balance->RemainingDays < $days) {
        return response()->json(["message" => "Insufficient leave balance"], 422);
    }

    $leave = LeaveRequest::create($validated + ["LeaveStatus" => "Pending"]);

    return response()->json($leave, 201);
}

public function approve(Request $request, int $id): JsonResponse
{
    $request->validate(["LeaveStatus" => "required|in:Approved,Rejected"]);

    $leave = LeaveRequest::findOrFail($id);
    $leave->update([
        "LeaveStatus" => $request->LeaveStatus,
        "ApprovedBy"  => $request->user()->EmployeeID,
    ]);

    if ($request->LeaveStatus === "Approved") {
        $days = Carbon::parse($leave->StartDate)
                      ->diffInWeekdays(Carbon::parse($leave->EndDate)) + 1;

        LeaveBalance::where("EmployeeID", $leave->EmployeeID)
                    ->where("LeaveType",  $leave->LeaveType)
                    ->increment("UsedDays", $days);
    }

    return response()->json($leave->fresh());
}
```

LEARNING NOTE — Carbon is Laravel's built-in date library. diffInWeekdays() counts working days between two dates, excluding weekends — appropriate for leave calculations. The after_or_equal validation rule replaces a manual date comparison. increment("UsedDays", $days) runs a single SQL UPDATE that adds to the existing value atomically, which is safer than reading the value, adding in PHP, and writing back (which could cause race conditions if two requests run simultaneously). The fresh() call at the end reloads the model from the database so the response reflects the saved state.

7 Attendance Tracking

AttendanceController.php

How it works

The controller stores a single attendance record per employee per day. A unique constraint on (EmployeeID, AttendanceDate) in the migration prevents duplicate entries for the same day. The index method accepts date range filters to retrieve attendance for reporting and payroll purposes.

app/Http/Controllers/AttendanceController.php

```
public function store(Request $request): JsonResponse
{
    $validated = $request->validate([
        "EmployeeID"      => "required|exists:Employee,EmployeeID",
        "AttendanceDate"  => "required|date",
        "TimeIn"          => "nullable|date_format:H:i:s",
        "TimeOut"         => "nullable|date_format:H:i:s|after:TimeIn",
        "AttendanceStatus"=> "required|in:Present,Absent,Late,Half-Day",
    ]);

    $attendance = Attendance::updateOrCreate(
        [
            "EmployeeID"     => $validated["EmployeeID"],
            "AttendanceDate" => $validated["AttendanceDate"],
        ],
        $validated
    );

    return response()->json($attendance, 201);
}

public function index(Request $request, int $employeeId): JsonResponse
{
    $records = Attendance::where("EmployeeID", $employeeId)
        ->when($request->from, fn($q) => $q->whereDate("AttendanceDate", ">=", $request->from))
        ->when($request->to,   fn($q) => $q->whereDate("AttendanceDate", "<=", $request->to))
        ->orderBy("AttendanceDate", "desc")
        ->paginate(30);

    return response()->json($records);
}
```

LEARNING NOTE — updateOrCreate() is one of Eloquent's most useful methods. The first array is the search criteria (find an attendance record matching this employee and date). The second array is what to write. If the record exists it is updated; if it does not exist it is created. This makes the endpoint idempotent — calling it twice with the same data has the same result as calling it once. The when() method is a conditional chain — the closure only runs if the first argument is truthy, keeping the query readable without if statements.

8 Payroll Processing

PayrollController.php · PayrollService.php

How it works

Payroll calculation is delegated to a PayrollService class to keep the controller thin. The service fetches all active AssignedAllowances and AssignedDeductions for the employee in the given period, sums them, and computes the NetSalary. The result is stored in the Payroll table with junction records in PayrollAllowance and PayrollDeduction linking each applied allowance and deduction.

app/Services/PayrollService.php

```
class PayrollService
{
```

```
    public function calculate(Employee $employee, string $payPeriod): Payroll
    {
        $today = now();

        $allowances = AssignedAllowance::where("EmployeeID", $employee->EmployeeID)
            ->where("EffectiveDate", "<=", $today)
            ->where(fn($q) => $q->whereNull("EndDate")->orWhere("EndDate", ">=", $today))
            ->get();

        $deductions = AssignedDeduction::where("EmployeeID", $employee->EmployeeID)
            ->where("EffectiveDate", "<=", $today)
            ->where(fn($q) => $q->whereNull("EndDate")->orWhere("EndDate", ">=", $today))
            ->get();

        $totalAllowances = $allowances->sum("Amount");
        $totalDeductions = $deductions->sum("Amount");
        $net = $employee->department->BasicSalary + $totalAllowances - $totalDeductions;

        $payroll = Payroll::create([
            "EmployeeID"  => $employee->EmployeeID,
            "PayPeriod"   => $payPeriod,
            "BasicSalary" => $employee->department->BasicSalary ?? 0,
            "NetSalary"   => max(0, $net),
            "PaymentDate" => now(),
        ]);

        foreach ($allowances as $a) {
            PayrollAllowance::create([
                "PayrollID" => $payroll->PayrollID,
                "AssignedAllowanceID" => $a->AssignedAllowanceID,
            ]);
        }

        foreach ($deductions as $d) {
            PayrollDeduction::create([
                "PayrollID" => $payroll->PayrollID,
                "AssignedDeductionID" => $d->AssignedDeductionID,
            ]);
        }

        return $payroll->load("allowances.allowance", "deductions.deduction");
    }
}
```

LEARNING NOTE — Putting this logic in PayrollService instead of the controller is called the Single Responsibility Principle. The controller only handles HTTP (receive request, return response). The service handles business rules. This makes the calculation independently testable — you can write a PHPUnit test that calls PayrollService->calculate() directly without making an HTTP request. The max(0, $net) call prevents a negative NetSalary if deductions exceed salary, which the database CHECK constraint would otherwise reject.

9 Audit Logging

`AuditLog model · Eloquent Observers`

How it works

Laravel Model Observers listen for Eloquent events (created, updated, deleted) and fire automatically when a model is saved or changed. By registering an observer on the Employee, LeaveRequest, AdditionalDocuments, and other key models, every database change is captured without writing any logging code in the controllers. The observer writes a row to the AuditLog table including a JSON snapshot of the data before and after the change.

Terminal — generate an observer

```
sail artisan make:observer EmployeeObserver --model=Employee
```

app/Observers/EmployeeObserver.php

```
class EmployeeObserver
{
    private function log(string $action, Employee $model, array $old = []): void
    {
        AuditLog::create([
            "UserID"          => auth()->id(),
            "Username"        => auth()->user()?->Username ?? "system",
            "Action"          => $action,
            "AffectedTable"   => "Employee",
            "AffectedRecordID"=> $model->EmployeeID,
            "OldValues"       => $old ?: null,
            "NewValues"       => $model->toArray(),
            "IPAddress"       => request()->ip(),
        ]);
    }

    public function created(Employee $employee): void
    {
        $this->log("CREATE", $employee);
    }

    public function updated(Employee $employee): void
    {
        $this->log("UPDATE", $employee, $employee->getOriginal());
    }
}
```

app/Providers/AppServiceProvider.php — register the observer

```
public function boot(): void
{
    Employee::observe(EmployeeObserver::class);
    LeaveRequest::observe(LeaveObserver::class);
    AdditionalDocuments::observe(DocumentObserver::class);
}
```

LEARNING NOTE — getOriginal() returns the attribute values as they were when the model was last loaded from the database, before any changes were applied. This gives you the "before" snapshot automatically — Laravel tracks this internally on every model. Observers are registered once in AppServiceProvider and then fire on every save across the entire application — you never need to call the logging code manually in any controller. The AuditLog table has no foreign keys intentionally — if the employee record is later deactivated, the audit entry must still be readable.

10 Notifications and Expiry Alerts

Notifications · Reverb · Mailgun · Redis Queue

How it works

Laravel's notification system sends the same notification through multiple channels simultaneously. A notification class defines what is sent via database (stored in the Notifications table) and via mail (dispatched through Mailgun). Notifications are dispatched onto a Redis queue so they do not slow down the HTTP response. A scheduled artisan command runs daily to check AdditionalDocuments for records where ExpiryDate is within 30 days and dispatches alerts. Real-time delivery to the browser uses Laravel Reverb via WebSocket events.

Terminal — create a notification class

```
sail artisan make:notification DocumentExpiryAlert
```

app/Notifications/DocumentExpiryAlert.php

```
class DocumentExpiryAlert extends Notification implements ShouldQueue
{
    use Queueable;

    public function __construct(
        private AdditionalDocuments $document
    ) {}

    public function via(object $notifiable): array
    {
        return ["database", "mail"];
    }

    public function toDatabase(object $notifiable): array
    {
        return [
            "Title"            => "Document expiring soon",
            "Message"          => "Document {$this->document->Description} "
                                  . "expires on {$this->document->ExpiryDate}",
            "NotificationType" => "EXPIRY_ALERT",
            "ReferenceTable"   => "AdditionalDocuments",
            "ReferenceID"      => $this->document->DocumentID,
        ];
    }

    public function toMail(object $notifiable): MailMessage
    {
        return (new MailMessage)
            ->subject("Document Expiry Alert — GardaWorld HRMS")
            ->line("A document is expiring within 30 days.")
            ->line("Employee: " . $this->document->employee->FullName)
            ->line("Document: " . $this->document->Description)
            ->line("Expiry Date: " . $this->document->ExpiryDate);
    }
}
```

routes/console.php — daily scheduled expiry check

```
Schedule::call(function () {
    $expiring = AdditionalDocuments::with(["employee.userAccount"])
        ->whereNotNull("ExpiryDate")
        ->whereBetween("ExpiryDate", [now(), now()->addDays(30)])
        ->get();

    foreach ($expiring as $doc) {
        $user = $doc->employee?->userAccount;
        if ($user) {
            $user->notify(new DocumentExpiryAlert($doc));
        }
    }
})->daily()->name("check-document-expiry");
```

LEARNING NOTE — implements ShouldQueue tells Laravel to dispatch this notification onto the Redis queue instead of sending it synchronously during the HTTP request. The queue worker (sail artisan queue:work) picks it up in the background and sends the email through Mailgun and writes the database row. This means the API response returns instantly and the notification is sent asynchronously. The via() method returning ["database", "mail"] means both channels fire — one class handles both. For the real-time browser notification, you broadcast an event via Reverb that the React frontend listens to using Laravel Echo.

11 Guard Deployment History

`DeploymentController.php`

How it works

The DeploymentController records each time a security guard is assigned to a site or branch. It validates that the EndDate is either null (still deployed) or after the StartDate. The index method can filter by employee or branch, enabling a branch manager to see all guards currently active at their site by filtering for records where EndDate is null.

app/Http/Controllers/DeploymentController.php

```
public function store(Request $request): JsonResponse
{
    $validated = $request->validate([
        "EmployeeID"     => "required|exists:Employee,EmployeeID",
        "BranchID"       => "nullable|exists:Branch,BranchID",
        "DeploymentSite" => "required|string|max:255",
        "StartDate"      => "required|date",
        "EndDate"        => "nullable|date|after_or_equal:StartDate",
        "Reason"         => "nullable|string",
    ]);

    $deployment = DeploymentHistory::create(
        $validated + ["DeployedBy" => $request->user()->EmployeeID]
    );

    return response()->json($deployment->load("employee", "branch"), 201);
}

public function currentlyDeployed(int $branchId): JsonResponse
{
    $guards = DeploymentHistory::with("employee")
        ->where("BranchID", $branchId)
        ->whereNull("EndDate")
        ->get();

    return response()->json($guards);
}
```

LEARNING NOTE — whereNull("EndDate") is the Eloquent equivalent of WHERE EndDate IS NULL in SQL. This retrieves only deployments that have not ended — i.e. guards currently on site. The $validated + ["DeployedBy" => ...] syntax merges the validated fields with the DeployedBy value which comes from the authenticated user, not the request body. This prevents a user from claiming someone else authorised the deployment.

12 User Account Management

`UserAccountController.php`

How it works

HR Administrators create, update, activate, deactivate, and reset passwords for user accounts. The controller always hashes the password before storing — plain text is never saved. Password resets generate a temporary password and email it to the user via Mailgun. Deactivation sets AccountStatus to inactive without deleting any records.

app/Http/Controllers/UserAccountController.php

```
public function store(Request $request): JsonResponse
{
    $validated = $request->validate([
        "EmployeeID" => "nullable|exists:Employee,EmployeeID|unique:UserAccount,EmployeeID",
        "Username"   => "required|string|unique:UserAccount,Username",
        "password"   => "required|string|min:8|confirmed",
        "RoleID"     => "required|exists:Role,RoleID",
    ]);
```

```
    $account = UserAccount::create([
        "EmployeeID"    => $validated["EmployeeID"] ?? null,
        "Username"      => $validated["Username"],
        "PasswordHash"  => Hash::make($validated["password"]),
        "RoleID"        => $validated["RoleID"],
        "AccountStatus" => "active",
    ]);

    return response()->json($account->load("role", "employee"), 201);
}

public function deactivate(int $id): JsonResponse
{
    UserAccount::findOrFail($id)->update(["AccountStatus" => "inactive"]);
    return response()->json(["message" => "Account deactivated"]);
}

public function resetPassword(Request $request, int $id): JsonResponse
{
    $request->validate(["password" => "required|string|min:8|confirmed"]);
    UserAccount::findOrFail($id)->update([
        "PasswordHash" => Hash::make($request->password),
    ]);
    return response()->json(["message" => "Password reset successfully"]);
}
```

LEARNING NOTE — The confirmed validation rule requires a matching password_confirmation field in the request body. This is a built-in Laravel convention — you do not write any extra code for it. Hash::make() is the correct way to hash a new password; Hash::check() is for verifying it later. Never use md5() or sha1() for passwords — they are not safe. bcrypt (which Hash::make() uses internally) is a slow hash function intentionally, making brute-force attacks computationally expensive.

13 Performance Evaluation and Training

`PerformanceEvaluationController · TrainingController`

How it works

Performance evaluations follow a straightforward store/index/show pattern. Training enrolment uses Eloquent's sync() and attach() methods on the many-to-many relationship between Employee and Training through the EmployeeTraining pivot table.

app/Http/Controllers/TrainingController.php — enrol an employee

```
public function enrol(Request $request, int $employeeId): JsonResponse
{
    $request->validate([
        "TrainingID"       => "required|exists:Training,TrainingID",
        "CompletionStatus" => "required|in:Enrolled,Completed,Failed",
    ]);

    $employee = Employee::findOrFail($employeeId);

    $employee->trainings()->syncWithoutDetaching([
        $request->TrainingID => [
            "CompletionStatus" => $request->CompletionStatus,
        ],
    ]);

    return response()->json($employee->load("trainings"), 200);
```

```
}
```

LEARNING NOTE — syncWithoutDetaching() adds or updates the pivot record for the given Training without removing existing enrolments. It is the correct choice when adding a single enrolment. sync() without "WithoutDetaching" would remove all other enrolments not in the array — use that only when replacing the entire enrolment list. The second argument to syncWithoutDetaching is an array of extra pivot columns (CompletionStatus in this case) that get stored in the EmployeeTraining junction table.

rder Recommendation

Implement features in this sequence for a workin

1st	**Authentication (F1) + User Account Management (F12)**	Nothing else works without this. Build login, logout, and me first. Confirm with Postman before moving on.
2nd	**Role Middleware (F2)**	Protect all routes immediately after auth works. Test with an Auditor account trying to hit an Admin-only route.
3rd	**Employee Profile (F3) + Search (F5)**	Core of the system. Seed a few employees and verify search filters work.
4th	**Document Upload (F4)**	Requires Backblaze B2 config. Test upload and temporary URL generation.
5th	**Audit Logging (F9)**	Register observers. Verify that every employee create/update writes to AuditLog.
6th	**Leave (F6) + Attendance (F7)**	Common HR workflows. Test the full approve cycle with two user accounts.
7th	**Deployment History (F8)**	Security-specific. Test the currentlyDeployed filter for a branch.
8th	**Notifications (F10)**	Run queue:work in a second terminal. Trigger an expiry and verify email arrives via Mailgun.
9th	**Payroll (F8) + Training (F13) + Recruitment (F11)**	Complex modules. Build after the core is stable and tested.

After completing all features in the backend, your Postman collection becomes the full API contract for the React frontend. Every endpoint, request format, and response shape documented in Section 2 maps directly to a fetch() or axios call in your React components.