Lab Project: Enterprise-Grade URL Shortener Microservice

Modules Covered: 5, 6, 7, 8, & 9 Integration

Framework: Django 5.0+ with Django REST Framework (DRF)

Timeline: 15 Working Days (Estimated)

---

1. Project Overview

1.1 Project Description

You will build a production-grade URL shortening service similar to Bitly or TinyURL designed as a highly scalable **microservice**. Unlike simple "To-Do" apps, this project simulates a real-world software engineering environment. You will start by building a clean, Dockerized foundation (Module 5), and progressively evolve it into a complex system handling relational data (Module 6), advanced security (Module 7), high-performance background tasks (Module 8), and distributed service communication (Module 9).

1.2 Core Project Objectives

By completing this project, you will demonstrate mastery of the following:

1. **Foundational Architecture (Module 5):** Setting up a professional Django project structure, containerization with Docker, and adhering to RESTful API standards.
2. **Data Engineering (Module 6):** Designing complex database schemas, optimizing queries, and mastering the Django ORM.
3. **Security & Access Control (Module 7):** Implementing industry-standard JWT authentication and granular Role-Based Access Control (RBAC).
4. **Performance Optimization (Module 8):** utilizing Redis for caching and Celery for asynchronous background processing to handle high traffic.
5. **Microservices Patterns (Module 9):** Implementing API Gateway patterns, resilient HTTP communication, and service decoupling.

---

2. System Architecture & Data Design

3.1 Database Schema (Module 6 Focus)

The integrity of your application relies on a strictly defined database schema. You must implement the following models with at least these specified field constraints but not limited to these only (you can add more fields if you need to).

A. User Model

- **Inheritance:** Extend Django's `AbstractUser`.
- **Fields:**
  - `email`: EmailField (Unique, Required).
  - `is_premium`: BooleanField (Default: False). Used to trigger premium logic.
  - `tier`: CharField (Choices: 'Free', 'Premium', 'Admin').

B. URL Model

- **Purpose:** Stores the mapping between short codes and original long URLs.
- **Fields:**
  - `original_url`: URLField (Must validate standard URL formats).
  - `short_code`: CharField (Unique, Indexed, Max Length 10). *Crucial for lookup performance.*
  - `custom_alias`: CharField (Nullable, Unique). Allows premium users to set vanity URLs.
  - `owner`: ForeignKey to User (ON DELETE CASCADE).
  - `is_active`: BooleanField (Default: True).
  - `expires_at`: DateTimeField (Nullable).
  - `title`: CharField(Nullable, Unique).
  - `description`: CharField(Nullable, Unique).
  - `favicon`: CharField(Nullable, Unique).
  - `click_count`: PositiveIntegerField (Default: 0). Denormalized counter for quick read access.

C. Click (Analytics) Model

- **Purpose:** Logs every visit to a short link for analytics.

- **Fields:**
    - `url`: ForeignKey to URL (ON DELETE CASCADE).
    - `clicked_at`: DateTimeField (Auto Now Add).
    - `ip_address`: GenericIPAddressField.
    - `city`: CharField (Nullable).
    - `country`: CharField (Nullable).
    - `user_agent`: TextField (Stores browser/OS info).
    - `referrer`: URLField (Nullable, tracks where the click came from).

D. Tag Model

- **Purpose:** Categorization for URLs.
- **Fields:** `name` (CharField, Unique).
- **Relationship:** Many-to-Many with `URL` model.

---

3. Technical Implementation by Module

Module 5: Fundamentals, Architecture & Containerization

Goal: Establish the project foundation, Docker environment, and a Minimum Viable Product (MVP).

What this module adds to the project:
In this phase, you are not concerned with users or analytics yet. The focus is purely on **infrastructure** and **core logic**. You will create a "clean" application that can accept a long URL, generate a unique short code, and redirect users. You will also containerize the application to ensure it runs identically on every machine.

Technical Implementation (Django Focus):

- **Project Structure:**
    - Initialize a Django project using a modular structure (e.g., separate apps for core, shortener, and api).
    - **Configuration Management:** Use python-decouple or django-environ to separate settings (SECRET_KEY, DEBUG, DATABASE_URL) from the code.
- **Core Logic (MVP):**
    - Create a basic URL model with original_url and short_code.

    - Implement a helper function/service to generate random 6-character alphanumeric strings for the short_code.
- **REST API (DRF):**
    - Implement POST /api/urls/ to accept a URL and return the short code.
    - Implement GET /<short_code>/ to redirect to the original URL (using HTTP 302).
    - Use **DRF Serializers** for strict data validation (ensure URLs are valid).
- **Containerization:**
    - **Dockerfile:** Write a multi-stage Dockerfile to build a lightweight Python image.
    - **Docker Compose:** Create a docker-compose.yml that spins up the Django web service and a Postgres database.
- **Documentation:**
    - Integrate drf-yasg or drf-spectacular to auto-generate Swagger/OpenAPI documentation.

Module 6: ORM & Data Access Layer

Goal: Expand the data model to support users, relationships, and deep analytics.

What this module adds to the project:
Now that the core works, you need to make it "smart." You will add user ownership (who created this link?), tagging (categorization), and analytics (who clicked this link?). This module focuses heavily on how to fetch this data efficiently without crashing the database.

Technical Implementation:

- **Advanced Modeling:**
    - **User Model:** Extend AbstractUser to add fields like is_premium and tier.
    - **URL Model Enhancements:** Add owner (ForeignKey), click_count, is_active, and expires_at.
    - **Click Model:** Create a model to log every visit: ip_address, user_agent, country, city, referer.
    - **Relationships:** Implement Many-to-Many relationships for **Tags** (e.g., "Marketing", "Social").
- **Database Migrations:**
    - Manage schema changes using proper Django migrations.
    - *Requirement:* Create a data migration to seed initial default tags.

- **Custom Managers & QuerySets:**
  - Implement URLManager with methods: active_urls(), expired_urls(), and popular_urls().
- **Query Optimization (Crucial):**
  - **N+1 Problem:** Use select_related when fetching URLs with their Owners. Use prefetch_related when fetching URLs with their Tags.
  - **Indexing:** Add database indexes to short_code (for fast lookups) and created_at.
  - **Aggregation:** Use annotate() to calculate complex stats (e.g., "Total clicks per country") directly in the database, not in Python.

Module 7: Authentication & Authorization

Goal: Secure the application and implement business rules based on user roles.

What this module adds to the project:

Currently, anyone can shorten a link. You will now lock down the API so only registered users can create links. You will also implement a **business model**: Free users get basic features, while Premium users get advanced features (like custom aliases).

Technical Implementation:

- **JWT Authentication:**
  - Install djangorestframework-simplejwt.
  - Implement endpoints for Login, Register, and Token Refresh.
- **Role-Based Access Control (RBAC):**
  - **Custom Permissions:** Create a permission class IsOwnerOrReadOnly (users cannot edit/delete other people's links).
  - **Tiered Logic:**
    - **Free Users:** Max 10 active URLs. No custom aliases (e.g., bit.ly/my-shop).
    - **Premium Users:** Unlimited URLs. Can specify custom aliases. Access to detailed analytics.
- **Security Best Practices:**
  - Implement **Rate Limiting** on the Login endpoint (e.g., 5 attempts per minute).
  - Store passwords securely (Django defaults) and sanitize all URL inputs.

Module 8: Advanced Optimization & Production Readiness

Goal: Prepare the system for high traffic using caching and background workers.

What this module adds to the project:
In a real URL shortener, writing to the database every time someone clicks a link is too slow. This module introduces **Async Tasks** (to handle analytics in the background) and **Caching** (to make redirects instant).

Technical Implementation:

- **Redis Caching:**
  - **Redirect Strategy:** The GET /<short_code> endpoint must check Redis first. If the URL is found, redirect immediately. If not, fetch from DB and cache it.
  - **Invalidation:** If a user updates a URL, delete the cache key immediately.
- **Asynchronous Tasks (Celery):**
  - **The "Write-Behind" Pattern:** When a link is clicked, do **not** write to the Click model in the view. Instead, trigger a Celery task: track_click_task.delay(url_id, ip_address, user_agent). This keeps the response time fast.
  - **Periodic Tasks (Celery Beat):** Configure a nightly task to "clean up" or archive expired URLs.
- **Logging & Monitoring:**
  - Configure structured logging (JSON format preferred) for all 500 errors and security warnings.
  - Add a /health endpoint to check if DB and Redis are reachable.

Module 9: Microservices Essentials

Goal: Simulate a distributed architecture and external service integration.

What this module adds to the project:
Microservices rarely live in isolation. This module adds a feature that requires your app to "talk" to another service. You will implement a "URL Preview" feature that fetches the title, description and favicon of the destination website.

Technical Implementation:

- **External Service Integration:**

    - **Scenario:** When a user submits a link to be shortened, your system should fetch the (title, description, favicon) of that page and store them in the database.

    - **Implementation:** Create a Service Layer function that uses httpx or requests to call the target URL.
        - Create a separate minimal Django app representing this "external" service (**URL Preview service**).
        - **Inter-Service Communication:** When a user creates a Short URL, your main app must send an HTTP request (using `httpx` or `requests`) to the Preview Service to fetch the page title,description, and favicon then store them in the **URL Model**.

- **Resiliency Patterns:**
    - **Retries:** If the target website is down, the request should fail gracefully or retry with exponential backoff.
    - **Circuit Breaker (Bonus):** If a target domain keeps failing, stop trying to fetch titles from it for a while.
- **API Gateway Concepts:**
    - Standardize your API versioning (e.g., /api/v1/...).
    - Document how a frontend (like React) would interact with your service via CORS headers.

---

4. API Endpoint Specifications

Your application must expose the following RESTful endpoints.

Authentication

- `POST /api/v1/auth/register/` - Create new account.
- `POST /api/v1/auth/login/` - Returns Access/Refresh JWT.
- `POST /api/v1/auth/refresh/` - Get new Access token.

URL Operations

- `POST /api/v1/urls/` - Create a short link. (Check rate limits/quotas here).
- `GET /api/v1/urls/` - List my URLs. (Supports pagination, search by tag).
- `GET /api/v1/urls/{short_code}/` - Retrieve details of a specific URL.

- `PUT /api/v1/urls/{short_code}/` - Update target URL (Reset click count optional).
- `DELETE /api/v1/urls/{short_code}/` - Deactivate/Delete URL.

Public Interface

- `GET /{short_code}/` - **The Redirection Endpoint**.
    - *Implementation Note:* Returns `302 Found` or `301 Moved Permanently`.
    - *Must trigger:* Async analytics task + Redis cache lookup.

Analytics

- `GET /api/v1/analytics/{short_code}/` - Detailed stats for one URL.
    - *Premium Only:* Returns time-series data and geo-location breakdown.

---

5. Project Roadmap & Milestones

The timeline assumes 15 working days to accommodate the addition of Module 5 foundation work.

Phase	Duration	Module	Key Deliverables
Phase 1: Foundation	Days 1-3	**Mod 5**	Docker setup, Django Project Init, Basic Create/Redirect endpoints, Swagger Docs.
Phase 2: Data Logic	Days 4-6	**Mod 6**	User/Click models, Relationships, Migrations, Managers, N+1 Optimization.
Phase 3: Security	Days 7-9	**Mod 7**	JWT Setup, Custom Permissions, Free vs Premium logic, Rate Limiting.

Phase 4: Scale	Days 10-12	**Mod 8**	Redis Cache for redirects, Celery setup, Async click tracking.
Phase 5: Integration	Days 13-15	**Mod 9**	External HTTP calls (Title fetch), Final QA, Documentation, Demo Video.

---

6. Technical Stack Requirements

You must use the following stack to ensure compatibility with grading tools:

Category	Technology
Language	Python 3.11+
Framework	Django 5.0+ & Django REST Framework
Database	PostgreSQL 15+ (psycopg2-binary)
Auth	djangorestframework-simplejwt
Caching	Redis 7+ (django-redis)
Async Tasks	Celery 5+ (with Redis as Broker)
Containerization	Docker & Docker Compose

Documentation	drf-yasg or drf-spectacular
WSGI Server	Gunicorn

---

7. Submission Guidelines

1. **GitHub Repository:** Must include a clear .gitignore, requirements.txt, and atomic commits.

2. **README.md:** A professional document explaining:
    - How to run the project (Docker commands).
    - The Architecture Diagram.
    - List of Endpoints.

Good luck! Focus on writing clean, modular code that you would be proud to deploy to production.