Feature Flag System - Complete Documentation

Overview

A production-ready, database-driven feature flag system for dynamic feature control without redeployment.

Tech Stack: Node.js, Express, TypeScript, MongoDB, Redis

Key Features:

- ☑ Global enable/disable
- ☑ Role-based access (ADMIN, PHARMACIST, CUSTOMER)
- ☑ User whitelist
- ☑ Gradual rollout (0-100%)
- ☑ Redis caching for performance
- ☑ Cache invalidation on updates
- ☑ Zero downtime feature deployment

Folder Structure

```
PHRMA-PRODUCTION-APP-BACKEND-MAIN/
│
├── Databases/
│   ├── Entities/
│   │   └── featureFlag.Interface.ts      # TypeScript interface
│   ├── Schema/
│   │   └── featureFlag.Schema.ts         # Mongoose schema
│   └── Models/
│       └── featureFlag.Models.ts         # Mongoose model
│
├── Services/
│   └── featureFlag.Service.ts            # Business logic + caching
│
├── Middlewares/
│   └── featureFlagMiddleware.ts          # requireFeature() middleware
│
├── Routers/
│   ├── Routers/
│   │   ├── featureFlag.Routes.ts         # Admin CRUD routes
│   │   └── features.Routes.ts            # Public user routes
│   └── main.Routes.ts                    # Main router (updated)
│
├── Utils/
│   └── Roles.enum.ts                     # Updated with PHARMACIST role
│
├── config/
```

```
│   └── redis.ts                          # Redis connection (existing)
│
└── examples/
    └── featureFlag.examples.ts           # Usage examples
```

MongoDB Schema

```
{
  key: String,              // "ONLINE_PAYMENT", "FEATURED_MEDICINES"
  name: String,             // "Online Payment Gateway"
  description: String,      // Optional description
  enabled: Boolean,         // Global on/off switch
  allowedRoles: [String],   // ["ADMIN", "PHARMACIST", "CUSTOMER"]
  allowedUserIds: [ObjectId], // User whitelist
  rolloutPercentage: Number, // 0-100
  createdAt: Date,
  updatedAt: Date
}
```

Indexes:

- `key` (unique, uppercase)
- `{ enabled: 1, key: 1 }`
- `{ allowedRoles: 1 }`

Request Flow

1. **Feature Check Request**

```
User Request → Auth Middleware → requireFeature("FEATURE_KEY")
```

2. **Feature Evaluation Logic**

```
1. Check Redis cache for feature flag
2. If not cached → Fetch from MongoDB
3. Populate Redis cache (TTL: 1 hour)
4. Evaluate feature for user:
   a. Is feature globally enabled?
   b. Is user in whitelist? (bypasses all other checks)
   c. Is user role allowed?
   d. Does user fall within rollout percentage?
5. Return true/false
```

```
6. If false → Return 403 Forbidden
7. If true → Proceed to controller
```

3. **Cache Invalidation**

```
Update/Delete Feature Flag → Invalidate Redis → Force fresh DB read
```

API Endpoints

Admin Routes (Protected: Admin Only)

1. Create Feature Flag

```
POST /api/feature-flags
Authorization: Bearer <admin_token>

{
  "key": "ONLINE_PAYMENT",
  "name": "Online Payment Gateway",
  "description": "Enables online payment processing",
  "enabled": true,
  "allowedRoles": ["ADMIN", "CUSTOMER"],
  "allowedUserIds": [],
  "rolloutPercentage": 100
}
```

2. Get All Feature Flags

```
GET /api/feature-flags
Authorization: Bearer <admin_token>
```

3. Get Feature Flag by Key

```
GET /api/feature-flags/ONLINE_PAYMENT
Authorization: Bearer <admin_token>
```

4. Update Feature Flag

```
PUT /api/feature-flags/ONLINE_PAYMENT
Authorization: Bearer <admin_token>

{
  "enabled": false,
  "rolloutPercentage": 50
}
```

5. Delete Feature Flag

```
DELETE /api/feature-flags/ONLINE_PAYMENT
Authorization: Bearer <admin_token>
```

6. Bulk Update

```
POST /api/feature-flags/bulk-update
Authorization: Bearer <admin_token>

{
  "updates": [
    { "key": "ONLINE_PAYMENT", "enabled": true },
    { "key": "AI_CHATBOT", "enabled": false }
  ]
}
```

7. Clear Cache (Debug)

```
DELETE /api/feature-flags/cache/clear
Authorization: Bearer <admin_token>
```

Public Route (Protected: Any Authenticated User)

Get User's Enabled Features

```
GET /api/features
Authorization: Bearer <user_token>

Response:
{
  "success": true,
```

```
  "message": "User features retrieved successfully",
  "data": {
    "ONLINE_PAYMENT": true,
    "FEATURED_MEDICINES": false,
    "AI_CHATBOT": true,
    "ADVANCED_ANALYTICS": false
  }
}
```

💻 Usage in Code

Protecting Routes with Middleware

```
import { requireFeature } from '../Middlewares/featureFlagMiddleware';
import { customersMiddleware } from '../Middlewares/CheckLoginMiddleware';

// Example 1: Simple protection
router.get(
  '/featured-medicines',
  customersMiddleware,
  requireFeature('FEATURED_MEDICINES'), // ← Blocks if disabled
  getFeaturedMedicines
);

// Example 2: Payment processing
router.post(
  '/process-payment',
  customersMiddleware,
  requireFeature('ONLINE_PAYMENT'),
  processPayment
);

// Example 3: AI features
router.post(
  '/ai-chat',
  customersMiddleware,
  requireFeature('AI_CHATBOT'),
  handleAIChatRequest
);
```

Programmatic Feature Checks

```
import FeatureFlagService from '../Services/featureFlag.Service';

// In controller
const isEnabled = await FeatureFlagService.isFeatureEnabled(
  'ONLINE_PAYMENT',
```

```
  userId,
  userRole
);

if (isEnabled) {
  // Show payment button
}
```

Conditional Dashboard Example

```
router.get('/dashboard', customersMiddleware, async (req, res) => {
  const userId = req.user._id;
  const userRole = req.user.role;

  // Check features
  const hasPayment = await FeatureFlagService.isFeatureEnabled(
    'ONLINE_PAYMENT', userId, userRole
  );

  const dashboard = {
    widgets: hasPayment ? ['payment', 'orders'] : ['orders']
  };

  res.json(dashboard);
});
```

🩹 Business Case Example

"Featured Medicines" Feature

Scenario:

- Currently: ADMIN only
- Goal: Gradually roll out to CUSTOMER

Implementation Steps:

Step 1: Create Feature Flag

```
POST /api/feature-flags
{
  "key": "FEATURED_MEDICINES",
  "name": "Featured Medicines Section",
  "enabled": true,
  "allowedRoles": ["ADMIN"],
  "rolloutPercentage": 100
}
```

Step 2: Protect Route

```
router.get(
  '/featured-medicines',
  requireFeature('FEATURED_MEDICINES'),
  getFeaturedMedicines
);
```

Step 3: Test with Admin

- Admin users: ☑ Access granted
- Customer users: ❌ 403 Forbidden

Step 4: Gradual Rollout to Customers

```
PUT /api/feature-flags/FEATURED_MEDICINES
{
  "allowedRoles": ["ADMIN", "CUSTOMER"],
  "rolloutPercentage": 20
}
```

- Now 20% of customers can access

Step 5: Full Rollout

```
PUT /api/feature-flags/FEATURED_MEDICINES
{
  "rolloutPercentage": 100
}
```

- All customers can access

Step 6: Emergency Disable

```
PUT /api/feature-flags/FEATURED_MEDICINES
{
  "enabled": false
}
```

- Feature disabled for everyone instantly

🚀 Performance

Caching Strategy

- **First Request:** Redis miss → MongoDB read → Cache write (slow)
- **Subsequent Requests:** Redis hit → Instant response (<1ms)
- **Cache TTL:** 1 hour
- **Invalidation:** On create/update/delete

Load Test Results (Estimated)

- Cached requests: **~10,000 req/s**
- Uncached requests: **~500 req/s**
- Cache hit ratio: **>95%**

🔒 Security

Access Control

- Admin routes: `adminMiddleware`
- Public routes: Authenticated users only
- Feature evaluation: User context required

Fail-Safe Behavior

- Redis down: Fall back to MongoDB
- MongoDB down: Disable all features (fail closed)
- Error in evaluation: Deny access (403)

🛠️ Configuration

Environment Variables

```
# Redis connection (already configured)
REDIS_URL=redis://localhost:6379

# MongoDB connection (already configured)
MONGO_URI=mongodb://localhost:27017/pharma
```

Redis Connection

Located in `config/redis.ts`:

- Auto-reconnect on failure
- Max 5 retries

- Connection timeout: 5 seconds

📊 Rollout Percentage Logic

How It Works

```
// User ID is hashed to get consistent percentage bucket
hash(userId) % 100 = userPercentile

if (userPercentile < rolloutPercentage) {
  // User is in rollout group
}
```

Example

- Rollout: 30%
- User A: Hash = 25 → ☑ Enabled
- User B: Hash = 75 → ❌ Disabled
- User A will **always** be in 30% group (consistent)

🧩 Integration Steps

1. Import in Your Route

```
import { requireFeature } from '../Middlewares/featureFlagMiddleware';
```

2. Add Middleware to Route

```
router.get('/my-feature', requireFeature('MY_FEATURE'), controller);
```

3. Create Feature Flag via API

```
POST /api/feature-flags
{
  "key": "MY_FEATURE",
  "enabled": true,
  "allowedRoles": ["CUSTOMER"],
  "rolloutPercentage": 100
}
```

4. Done! ☑

Feature is now controlled dynamically.

🐛 Debugging

Check Cache

```
GET /api/feature-flags/FEATURE_KEY
```

Clear Cache

```
DELETE /api/feature-flags/cache/clear
```

Check User Features

```
GET /api/features
```

Logs

```
# Redis connection
☑ Connected to Redis

# Feature evaluation
Feature flag evaluation error for ONLINE_PAYMENT: <error>

# Cache operations
Cache read error for ONLINE_PAYMENT: <error>
Cache write error for ONLINE_PAYMENT: <error>
```

📝 Example Feature Flags to Create

```
[
  {
    "key": "ONLINE_PAYMENT",
    "name": "Online Payment Gateway",
    "enabled": true,
    "allowedRoles": ["ADMIN", "CUSTOMER"],
    "rolloutPercentage": 100
  },
```

```
    {
      "key": "FEATURED_MEDICINES",
      "name": "Featured Medicines Section",
      "enabled": true,
      "allowedRoles": ["ADMIN"],
      "rolloutPercentage": 100
    },
    {
      "key": "AI_CHATBOT",
      "name": "AI-Powered Chatbot",
      "enabled": false,
      "allowedRoles": ["ADMIN", "CUSTOMER"],
      "rolloutPercentage": 0
    },
    {
      "key": "ADVANCED_ANALYTICS",
      "name": "Advanced Analytics Dashboard",
      "enabled": true,
      "allowedRoles": ["ADMIN"],
      "rolloutPercentage": 100
    },
    {
      "key": "PRESCRIPTION_UPLOAD",
      "name": "Prescription Upload Feature",
      "enabled": true,
      "allowedRoles": ["ADMIN", "PHARMACIST", "CUSTOMER"],
      "rolloutPercentage": 50
    }
  ]
```

☑ Checklist for Deployment

- ☑ MongoDB schema created
- ☑ Redis connection configured
- ☑ Service layer with caching implemented
- ☑ Middleware for route protection created
- ☑ Admin CRUD routes added
- ☑ Public features route added
- ☑ Routes registered in main router
- ☑ TypeScript types defined
- ☑ Error handling implemented
- ☑ Cache invalidation strategy in place
- ☑ Documentation complete

🎉 Summary

You now have a **complete, production-ready feature flag system** that:

1. ☑ Controls features dynamically via database
2. ☑ Supports role-based access and gradual rollouts
3. ☑ Uses Redis caching for high performance
4. ☑ Provides admin APIs for feature management
5. ☑ Offers public API for frontend integration
6. ☑ Enables zero-downtime feature deployment
7. ☑ Fails safely on errors (deny access)
8. ☑ Includes comprehensive examples and documentation

Next Steps:

1. Start server
2. Create feature flags via admin API
3. Protect your routes with `requireFeature()`
4. Test with different users and roles
5. Monitor Redis cache performance

Questions? Refer to `examples/featureFlag.examples.ts` for more use cases.