MONASH University × UpScale × AUSTRALIA CHESS WORLD THE ART OF CHESS

# BigCommerce Plugin for ChessWorld SKU Management

## Code Review

Version 1.0


**Written by:**

Rui En Koe

Alex Chan

Ashley Warden

Liangdi Wang

Luqmaan Yurzaa

Wen Cheng Huong

# Document History and Version Control

| Date (dd/mm/yy) | Version | Author | Description |
|---|---|---|---|
| 29/09/2025 | 1.0 | Liangdi Wang | Initial Draft. |

MONASH University × UpScale × AUSTRALIA CHESS WORLD THE ART OF CHESS

# Contents

# 1. Executive Summary

## 1.1 Purpose of this Document

This document provides a comprehensive code review of the BigCommerce Plugin for ChessWorld SKU Management. The review focuses on:

- **Code Quality**: Assessing adherence to best practices, maintainability, and readability
- **Architecture**: Evaluating the overall system design and component organization
- **Security**: Identifying potential vulnerabilities and security improvements
- **Performance**: Analyzing efficiency and scalability considerations
- **Maintainability**: Reviewing documentation, testing, and long-term sustainability

This review is intended to help the development team improve code quality, reduce technical debt, and establish better practices for future development.

## 1.2 Review Methodology

The code review was conducted using the following approach:

1. **Static Analysis**: Examination of code structure, patterns, and organization
2. **Architecture Review**: Assessment of system design and component relationships
3. **Best Practices Evaluation**: Comparison against industry standards and TypeScript/React conventions
4. **Security Assessment**: Identification of potential security vulnerabilities
5. **Performance Analysis**: Review of efficiency and scalability considerations

## 1.3 Overall Assessment

**Strengths:**

- Well-structured Next.js application with clear separation of concerns
- Effective use of BigCommerce APIs and webhook integration
- Good component-based architecture for React frontend
- Comprehensive database abstraction layer
- Proper environment configuration management

**Areas for Improvement:**

- TypeScript configuration could be more strict
- Inconsistent error handling patterns
- Limited code documentation and inline comments
- Missing input validation in several API endpoints

- Potential performance issues with sequential API calls
- Database schema inconsistencies between documentation and implementation

**Overall Rating: B+ (Good with room for improvement)**

# 2. Architecture Analysis

## 2.1 Technology Stack Assessment

The project uses a modern and appropriate technology stack:

**Frontend:**

- **Next.js 12.3.3**: Appropriate choice for BigCommerce apps, though could be updated to a more recent version
- **React 18.2.0**: Current and well-supported version
- **BigCommerce Big Design**: Excellent choice for UI consistency with BigCommerce admin
- **TypeScript**: Good choice for type safety, but configuration needs improvement

**Backend:**

- **Next.js API Routes**: Appropriate for this scale of application
- **PostgreSQL with Neon**: Good choice for production deployment
- **JWT Authentication**: Standard and secure approach

**Recommendations:**

- Consider upgrading Next.js to version 13+ for improved performance and features
- Implement stricter TypeScript configuration
- Add API rate limiting and caching mechanisms

## 2.2 Project Structure Evaluation

The project follows a logical and maintainable structure:

```
├── components/      # React components (well-organized)
├── lib/             # Utility functions and business logic
├── pages/           # Next.js pages and API routes
├── types/           # TypeScript type definitions
├── test/            # Test files and mocks
└── docs/            # Documentation
```

**Strengths:**

- Clear separation between frontend components and backend logic
- Dedicated types directory for TypeScript definitions
- Logical grouping of API endpoints
- Comprehensive documentation directory

**Areas for Improvement:**

- Some components could be further modularized
- Missing shared utilities directory for common functions
- API routes could benefit from middleware organization

## 2.3 Database Design Review

**Current Implementation:** The database layer shows some inconsistencies between documentation and actual implementation:

**Documented Schema (from Technical Guide):**

- `users`, `stores`, `store_users` tables for authentication
- Bundle data stored in BigCommerce metafields

**Actual Implementation (from postgres.ts):**

- Authentication tables as documented
- Additional `bundles` and `bundle_links` tables that appear unused
- Inconsistency between metafield-based and database-based storage

**Recommendations:**

1. **Clarify Data Storage Strategy**: Decide whether to use BigCommerce metafields or local database for bundle data
2. **Remove Unused Code**: Clean up unused database methods and tables
3. **Update Documentation**: Ensure database schema documentation matches implementation
4. **Add Database Migrations**: Implement proper migration system for schema changes

# 3. Code Quality Assessment

## 3.1 TypeScript Configuration

**Current Configuration Issues:**

```
{
  "strict": false,  // Should be true for better type safety
  "target": "es5"   // Could be updated to es2020 or newer
}
```

**Recommendations:**

- Enable strict mode: `"strict": true`
- Update target to `"es2020"` or newer
- Add `"noUncheckedIndexedAccess": true` for safer array/object access
- Enable `"exactOptionalPropertyTypes": true`

## 3.2 Code Organization and Modularity

**Strengths:**

- Good separation of concerns between authentication, database, and business logic
- Reusable components with clear props interfaces
- Consistent naming conventions

**Areas for Improvement:**

**1. Component Duplication:** The `CreateBundleForm` component is quite large (150+ lines) and could be broken down:

```
// Current: Single large component
const CreateBundleForm = ({ availableSKUs, onSubmit, onCancel, initialData }) => {
  // 150+ lines of logic
};

// Recommended: Split into smaller components
const BundleFormHeader = ({ name, onNameChange }) => { /* ... */ };
const SKUSelector = ({ availableSKUs, onAdd }) => { /* ... */ };
const BundleItemsTable = ({ items, onQuantityChange, onRemove }) => { /* ... */ };
```

**2. Utility Functions:** Several utility functions are scattered across files and could be centralized:

```
// Create: lib/utils/validation.ts
export const validateBundleData = (bundle: BundleData) => { /* ... */
};
export const formatPrice = (price: number) => { /* ... */ };
export const calculateBundlePrice = (items: BundleItem[]) => { /* ...
*/ };
```

## 3.3 Error Handling and Validation

**Current Issues:**

**1. Inconsistent Error Handling:**

```
// Some endpoints use try-catch with generic errors
catch (error: any) {
  console.error('Error fetching bundles:', error);
  res.status(500).json({ message: 'Internal server error' });
}

// Others have minimal error handling
const { accessToken, storeHash } = await getSession(req); // Could
throw
```

**2. Missing Input Validation:**

```
// API endpoints lack input validation
export default async function handler(req: NextApiRequest, res:
NextApiResponse) {
  // No validation of request body or parameters
  const { productId } = req.query; // Could be undefined or invalid
}
```

**Recommendations:**

- Implement consistent error handling middleware
- Add input validation using libraries like Zod or Joi
- Create standardized error response formats
- Add proper logging instead of console.error

## 3.4 Performance Considerations

**Current Performance Issues:**

**1. Sequential API Calls:**

```
// In bundles/index.ts - inefficient sequential calls
for (const product of products) {
```

```
            const      {      data:      metafields      }      =      await
bc.get(`/catalog/products/${product.id}/metafields`);
  // Process each product individually
}
```

**2. Missing Caching:**

- No caching mechanism for frequently accessed data
- Repeated API calls for the same information

**Recommendations:**

- Implement parallel API calls using `Promise.all()`
- Add Redis or in-memory caching for product data
- Implement request deduplication
- Add pagination for large datasets

# 4. Security Review

## 4.1 Authentication and Authorization

**Strengths:**

- Proper JWT implementation with expiration
- Secure session management
- Environment variable protection for secrets

**Areas for Improvement:**

**1. Development Bypass Security:**

```
// In auth.ts - potential security risk
if (
    (process.env.DEV_BC_BYPASS === '1' || process.env.NODE_ENV === 'development') &&
  process.env.ACCESS_TOKEN &&
  process.env.STORE_HASH
) {
  return {
    accessToken: process.env.ACCESS_TOKEN,
    storeHash: process.env.STORE_HASH,
    user: { id: 1, email: 'dev@localhost', username: 'dev' },
  };
}
```

**Recommendations:**

- Add additional checks to ensure development bypass is never active in production
- Implement role-based access control
- Add request rate limiting
- Implement CSRF protection

## 4.2 Data Validation and Sanitization

**Current Issues:**

- Missing input validation on API endpoints
- No sanitization of user inputs
- Potential for injection attacks through unvalidated parameters

**Recommendations:**

```
// Add input validation schema
import { z } from 'zod';
```

```
const BundleCreateSchema = z.object({
  name: z.string().min(1).max(100),
  items: z.array(z.object({
    sku: z.string().min(1),
    quantity: z.number().min(1).max(1000)
  })),
  price: z.number().positive().optional()
});

// Use in API endpoints
const validatedData = BundleCreateSchema.parse(req.body);
```

## 4.3 Environment Configuration

**Current Implementation:**

- Good use of environment variables for sensitive data
- Proper separation of development and production configs

**Recommendations:**

- Add environment variable validation on startup
- Implement configuration schema validation
- Add secrets rotation mechanism

# 5. Maintainability

## 5.1 Component Reusability

**Current Issues:**

- Some components are tightly coupled to specific use cases
- Limited reusability across different contexts
- Missing shared component library

**Recommendations:**

**1. Create Shared Components:**

```
// components/shared/DataTable.tsx
interface DataTableProps<T> {
  data: T[];
  columns: ColumnDefinition<T>[];
  onRowAction?: (action: string, item: T) => void;
}

// components/shared/FormField.tsx
interface FormFieldProps {
  label: string;
  error?: string;
  required?: boolean;
  children: React.ReactNode;
}
```

**2. Extract Business Logic:**

```
// hooks/useBundleManagement.ts
export const useBundleManagement = () => {
  const [bundles, setBundles] = useState<Bundle[]>([]);
  const [loading, setLoading] = useState(false);

  const createBundle = useCallback(async (bundleData: BundleData) => {
    // Business logic here
  }, []);

  return { bundles, loading, createBundle };
};
```

## 5.2 Testing Coverage

**Current State:**

- Basic Jest configuration present
- Limited test coverage
- Some snapshot tests for components

**Recommendations:**

- Increase unit test coverage to at least 80%
- Add integration tests for API endpoints
- Implement end-to-end tests for critical user flows
- Add performance testing for bundle calculations

# 6. Specific Improvement Recommendations

## 6.1 High Priority Issues

### Issue 1: Database Schema Inconsistency

**Problem:** Mismatch between documented architecture and actual implementation
**Impact:** Confusion for new developers, potential data integrity issues
**Solution:**

```
-- Remove unused tables if using metafields approach
DROP TABLE IF EXISTS bundle_links;
DROP TABLE IF EXISTS bundles;

-- OR implement proper database-based storage
-- Update documentation to match chosen approach
```

### Issue 2: Missing Input Validation

**Problem:** API endpoints lack proper input validation
**Impact:** Security vulnerabilities, data corruption risks
**Solution:**

```
// Add validation middleware
import { z } from 'zod';

const validateRequest = (schema: z.ZodSchema) => {
      return    (req:    NextApiRequest,    res:    NextApiResponse,    next:
NextFunction) => {
    try {
      schema.parse(req.body);
      next();
    } catch (error) {
      res.status(400).json({ error: 'Invalid request data' });
    }
  };
};
```

### Issue 3: Performance Issues with Sequential API Calls

**Problem:** Inefficient sequential API calls in bundle fetching
**Impact:** Slow response times, poor user experience
**Solution:**

```
// Replace sequential calls with parallel processing
const metafieldPromises = products.map(product =>
  bc.get(`/catalog/products/${product.id}/metafields`)
```

```
);
const metafieldResults = await Promise.all(metafieldPromises);
```

## 6.2 Medium Priority Improvements

### Issue 1: TypeScript Configuration

**Problem:** Loose TypeScript configuration reduces type safety
**Solution:**

```
{
  "compilerOptions": {
    "strict": true,
    "noUncheckedIndexedAccess": true,
    "exactOptionalPropertyTypes": true,
    "target": "es2020"
  }
}
```

### Issue 2: Error Handling Standardization

**Problem:** Inconsistent error handling across the application
**Solution:**

```
// lib/errors.ts
export class AppError extends Error {
  constructor(
    public message: string,
    public statusCode: number = 500,
    public code?: string
  ) {
    super(message);
  }
}

// middleware/errorHandler.ts
export const errorHandler = (error: Error, req: NextApiRequest, res:
NextApiResponse) => {
  if (error instanceof AppError) {
    return res.status(error.statusCode).json({
      error: error.message,
      code: error.code
    });
  }

  console.error('Unexpected error:', error);
```

```
  res.status(500).json({ error: 'Internal server error' });
};
```

### Issue 3: Component Modularity

**Problem:** Large components that are difficult to maintain
**Solution:** Break down large components into smaller, focused components

## 6.3 Low Priority Enhancements

### Issue 1: Caching Implementation

**Solution:** Add Redis caching for frequently accessed data

### Issue 2: Monitoring and Logging

**Solution:** Implement structured logging and application monitoring

### Issue 3: API Documentation

**Solution:** Add OpenAPI/Swagger documentation for API endpoints