CODING STANDARDS

STOCKFELLOW

DEVOPPS

BRIGHTBYTE ENTERPRISES

DEMO 3

Name and Surname	**Student Number**
*Tinotenda Chirozvi	22547747
Diyaana Jadwat	23637252
Dean Ramsey	22599012
Naazneen Khan	22527533
Lubabalo Tshikila	22644106

Contents

1 Introduction

This document establishes coding standards and best practices for the Stockfellow fintech application. These standards ensure code consistency, maintainability, security, and quality across our React frontend and Spring Boot backend systems.

1.1 Key Objectives

- Maintain consistent code style across the team
- Ensure code readability and maintainability
- Implement security best practices for financial applications
- Facilitate effective code reviews and collaboration
- Reduce onboarding time for new team members

2 General Principles

2.1 Code Quality Principles

- **Readability:** Code should be self-documenting and easy to understand
- **Consistency:** Follow established patterns and conventions
- **Simplicity:** Prefer simple, clear solutions over complex ones
- **Security:** Always consider security implications in financial software
- **Performance:** Write efficient code, especially for transaction processing
- **Testability:** Write code that is easy to unit test and mock

2.2 Naming Conventions

- Use meaningful, descriptive names for variables, functions, and classes
- Avoid abbreviations unless they are widely understood
- Use consistent terminology throughout the codebase
- Follow language-specific naming conventions (camelCase for JavaScript, PascalCase for Java classes)

3 Git and Version Control

3.1 Branch Naming

```
feature/TICKET-123-user-authentication
bugfix/TICKET-456-transaction-validation
hotfix/TICKET-789-security-patch
release/v1.2.0
```

3.2 Commit Message Format

```
type(scope): short description

Longer description if needed

Fixes #123
```

Examples:

```
feat(auth): implement OAuth 2.0 authentication

fix(transactions): resolve decimal precision issue in calculations

docs(api): update user endpoint documentation
```

3.3 Pull Request Guidelines

- Create small, focused pull requests
- Include descriptive title and description
- Reference related tickets/issues
- Ensure all tests pass before requesting review
- Add reviewers and appropriate labels

4 Java/Spring Boot Backend Standards

4.1 Project Structure

```
src/
          main/
                java/
                      com/stockfellow/
                          config/          # Configuration classes
                          controller/      # REST controllers
                          dto/             # Data Transfer Objects
                          entity/          # JPA entities
                          exception/       # Custom exceptions
                          repository/      # Data access layer
                          service/         # Business logic
```

```
                        security/       # Security configurations
                        util/           # Utility classes
                resources/
                    application.yml
                    db/migration/       # Flyway migrations
            test/
```

4.2 Naming Conventions

- **Classes:** PascalCase (UserService, TransactionController)
- **Methods:** camelCase (findUserById, processTransaction)
- **Variables:** camelCase (userId, transactionAmount)
- **Constants:** UPPER_SNAKE_CASE (MAX_TRANSACTION_AMOUNT)
- **Packages:** lowercase (com.stockfellow.service)

4.3 Class Design Example

```java
// Good: Single responsibility, clear naming
@Service
@Transactional
@Slf4j
public class UserService {

    private final UserRepository userRepository;
    private final EmailService emailService;

    public UserService(UserRepository userRepository, EmailService
        emailService) {
        this.userRepository = userRepository;
        this.emailService = emailService;
    }

    public User createUser(CreateUserRequest request) {
        validateUserRequest(request);

        User user = User.builder()
            .email(request.getEmail())
            .firstName(request.getFirstName())
            .lastName(request.getLastName())
            .build();

        User savedUser = userRepository.save(user);
        emailService.sendWelcomeEmail(savedUser);

        log.info("User created successfully: {}", savedUser.getId());
        return savedUser;
    }

    private void validateUserRequest(CreateUserRequest request) {
        if (userRepository.existsByEmail(request.getEmail())) {
            throw new UserAlreadyExistsException("User with email
                already exists");
```

```
        }
    }
}
```

4.4 Controller Guidelines

```
@RestController
@RequestMapping("/api/v1/users")
@Validated
public class UserController {

    private final UserService userService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public ResponseEntity<UserResponse> createUser(@Valid @RequestBody
        CreateUserRequest request) {
        User user = userService.createUser(request);
        return ResponseEntity.ok(UserResponse.from(user));
    }

    @GetMapping("/{userId}")
    public ResponseEntity<UserResponse> getUser(@PathVariable @Valid
        @Positive Long userId) {
        User user = userService.findUserById(userId);
        return ResponseEntity.ok(UserResponse.from(user));
    }
}
```

4.5 Exception Handling

```
@ControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ErrorResponse handleUserNotFound(UserNotFoundException ex) {
        log.error("User not found: {}", ex.getMessage());
        return ErrorResponse.builder()
            .code("USER_NOT_FOUND")
            .message(ex.getMessage())
            .timestamp(Instant.now())
            .build();
    }
}
```

4.6 Entity Guidelines

```
@Entity
@Table(name = "users")
@Builder
```

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    @Email
    private String email;

    @Column(name = "first_name", nullable = false)
    private String firstName;

    @Column(name = "last_name", nullable = false)
    private String lastName;

    @CreationTimestamp
    @Column(name = "created_at")
    private Instant createdAt;

    @UpdateTimestamp
    @Column(name = "updated_at")
    private Instant updatedAt;
}
```

5 React Frontend Standards

5.1 Project Structure

```
src/
          components/          # Reusable UI components
                common/         # Generic components
                feature/        # Feature-specific components
          pages/              # Page components
          hooks/              # Custom React hooks
          services/           # API services
          utils/              # Utility functions
          types/              # TypeScript type definitions
          store/              # State management (Redux/Zustand)
          styles/             # Global styles and themes
          __tests__/          # Test files
```

5.2 Naming Conventions

- **Components:** PascalCase (UserProfile, TransactionList)
- **Files:** PascalCase for components, camelCase for utilities
- **Variables/Functions:** camelCase (userName, handleSubmit)
- **Constants:** UPPER_SNAKE_CASE (API_BASE_URL)

5.3 Component Guidelines

```tsx
// Use TypeScript for type safety
interface UserProfileProps {
  userId: string;
  onUpdate?: (user: User) => void;
}

export const UserProfile: React.FC<UserProfileProps> = ({ userId, onUpdate }) => {
  const [user, setUser] = useState<User | null>(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    const fetchUser = async () => {
      try {
        setIsLoading(true);
        const userData = await userService.getUserById(userId);
        setUser(userData);
      } catch (err) {
        setError('Failed to load user profile');
        console.error('Error fetching user:', err);
      } finally {
        setIsLoading(false);
      }
    };

    fetchUser();
  }, [userId]);

  const handleUpdateProfile = async (updatedData: Partial<User>) => {
    try {
      const updatedUser = await userService.updateUser(userId, updatedData);
      setUser(updatedUser);
      onUpdate?.(updatedUser);
    } catch (err) {
      setError('Failed to update profile');
    }
  };

  if (isLoading) return <LoadingSpinner />;
  if (error) return <ErrorMessage message={error} />;
  if (!user) return <NotFound />;

  return (
    <div className="user-profile">
      <h2>{user.firstName} {user.lastName}</h2>
      <ProfileForm user={user} onSubmit={handleUpdateProfile} />
    </div>
  );
};
```

5.4 Custom Hooks

```
// Custom hook for API calls
export const useApi = <T>(apiCall: () => Promise<T>) => {
  const [data, setData] = useState<T | null>(null);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);

  const execute = useCallback(async () => {
    try {
      setIsLoading(true);
      setError(null);
      const result = await apiCall();
      setData(result);
    } catch (err) {
      setError(err instanceof Error ? err.message : 'An error occurred
          ');
    } finally {
      setIsLoading(false);
    }
  }, [apiCall]);

  useEffect(() => {
    execute();
  }, [execute]);

  return { data, isLoading, error, refetch: execute };
};
```

5.5 Service Layer

```
// API service with error handling
class UserService {
  private readonly baseUrl = process.env.REACT_APP_API_BASE_URL;

  async getUserById(userId: string): Promise<User> {
    const response = await fetch(`${this.baseUrl}/users/${userId}`, {
      headers: {
        'Authorization': `Bearer ${getAuthToken()}`,
        'Content-Type': 'application/json',
      },
    });

    if (!response.ok) {
      throw new Error(`Failed to fetch user: ${response.statusText}`);
    }

    return response.json();
  }

  async updateUser(userId: string, userData: Partial<User>): Promise<
     User> {
    const response = await fetch(`${this.baseUrl}/users/${userId}`, {
      method: 'PUT',
      headers: {
        'Authorization': `Bearer ${getAuthToken()}`,
        'Content-Type': 'application/json',
      },
```

```
      body: JSON.stringify(userData),
    });

    if (!response.ok) {
      throw new Error(`Failed to update user: ${response.statusText}`);
    }

    return response.json();
  }
}

export const userService = new UserService();
```

6 Database Standards

6.1 Table Naming

- Use snake_case for table names (users, user_accounts, transaction_history)
- Use plural nouns for table names
- Prefix tables with module name if needed (auth_users, payment_transactions)

6.2 Column Naming

- Use snake_case for column names
- Use descriptive names (first_name, created_at, transaction_amount)
- Always include id, created_at, updated_at columns

6.3 Migration Guidelines

```
-- V1__Create_users_table.sql
CREATE TABLE users (
    id BIGSERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL UNIQUE,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_users_email ON users(email);
```

7 API Standards

7.1 RESTful Endpoints

```
GET    /api/v1/users              # Get all users
GET    /api/v1/users/{id}         # Get user by ID
POST   /api/v1/users              # Create new user
PUT    /api/v1/users/{id}         # Update user
DELETE /api/v1/users/{id}         # Delete user
```

7.2 Response Format

```
{
  "data": {
    "id": 1,
    "email": "user@example.com",
    "firstName": "John",
    "lastName": "Doe"
  },
  "timestamp": "2023-10-01T12:00:00Z",
  "success": true
}
```

7.3 Error Response Format

```
{
  "error": {
    "code": "USER_NOT_FOUND",
    "message": "User with ID 123 not found",
    "details": {}
  },
  "timestamp": "2023-10-01T12:00:00Z",
  "success": false
}
```

8 Testing Standards

8.1 Backend Testing

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @Mock
    private EmailService emailService;

    @InjectMocks
    private UserService userService;

    @Test
    void shouldCreateUserSuccessfully() {
        // Given
        CreateUserRequest request = CreateUserRequest.builder()
```

```
            .email("test@example.com")
            .firstName("John")
            .lastName("Doe")
            .build();

        User savedUser = User.builder()
            .id(1L)
            .email("test@example.com")
            .firstName("John")
            .lastName("Doe")
            .build();

        when(userRepository.existsByEmail(request.getEmail())).
           thenReturn(false);
        when(userRepository.save(any(User.class))).thenReturn(savedUser
           );

        // When
        User result = userService.createUser(request);

        // Then
        assertThat(result).isNotNull();
        assertThat(result.getEmail()).isEqualTo("test@example.com");
        verify(emailService).sendWelcomeEmail(savedUser);
    }
}
```

8.2 Frontend Testing

```
import { render, screen, fireEvent, waitFor } from '@testing-library/
   react';
import { UserProfile } from '../UserProfile';
import { userService } from '../../services/userService';

jest.mock('../../services/userService');

describe('UserProfile', () => {
  const mockUser = {
    id: '1',
    email: 'test@example.com',
    firstName: 'John',
    lastName: 'Doe',
  };

  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('should display user profile when loaded successfully', async ()
     => {
    // Given
    (userService.getUserById as jest.Mock).mockResolvedValue(mockUser);

    // When
    render(<UserProfile userId="1" />);

```

```
    // Then
    await waitFor(() => {
      expect(screen.getByText('John Doe')).toBeInTheDocument();
    });
  });

  it('should display error message when user fetch fails', async () => {
    // Given
    (userService.getUserById as jest.Mock).mockRejectedValue(
      new Error('User not found')
    );

    // When
    render(<UserProfile userId="1" />);

    // Then
    await waitFor(() => {
      expect(screen.getByText(/Failed to load user profile/)).
          toBeInTheDocument();
    });
  });
});
```

9 Documentation Standards

9.1 Code Comments

```
/**
 * Processes a financial transaction with validation and audit logging.
 *
 * @param transactionRequest the transaction details to process
 * @return the processed transaction with updated status
 * @throws InsufficientFundsException if account balance is
    insufficient
 * @throws InvalidAccountException if account is not found or inactive
 */
public Transaction processTransaction(TransactionRequest
   transactionRequest) {
    // Implementation
}
```

9.2 README Guidelines

Each module should include:

- Purpose and overview
- Setup and installation instructions
- Configuration requirements
- API documentation links

- Testing instructions
- Deployment guidelines

10 Security Standards

10.1 Input Validation

```
// Always validate input data
@PostMapping("/transfer")
public ResponseEntity<TransactionResponse> transfer(
    @Valid @RequestBody TransferRequest request
) {
    // Validate transaction amount
    if (request.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
        throw new InvalidTransactionException("Amount must be positive");
    }

    // Process transfer
    Transaction transaction = transactionService.processTransfer(request);
    return ResponseEntity.ok(TransactionResponse.from(transaction));
}
```

10.2 Sensitive Data Handling

```
// Never log sensitive data
log.info("Processing transaction for user: {}", userId); // Good
log.info("Processing transaction: {}", transaction.toString()); // Bad - may contain sensitive data

// Use proper encryption for sensitive fields
@Column(name = "account_number")
@Convert(converter = EncryptedStringConverter.class)
private String accountNumber;
```

10.3 Authentication & Authorization

```
@PreAuthorize("hasRole('USER') and #userId == authentication.principal.id")
@GetMapping("/users/{userId}/accounts")
public ResponseEntity<List<AccountResponse>> getUserAccounts(
    @PathVariable Long userId) {
    // Implementation
}
```

11 Code Review Guidelines

11.1 Review Checklist

☐ Code follows established patterns and conventions

☐ All tests pass and new tests are added for new functionality

☐ Security best practices are followed

☐ No sensitive data is logged or exposed

☐ Error handling is appropriate

☐ Documentation is updated if needed

☐ Code is readable and well-documented

11.2 Review Process

- **Self-review:** Author reviews their own code before requesting review
- **Peer review:** At least one team member reviews the code
- **Testing:** Ensure all automated tests pass
- **Security review:** Check for security vulnerabilities
- **Documentation:** Verify documentation is updated
- **Approval:** Code must be approved before merging

11.3 Review Comments

- Be constructive and specific
- Explain the reasoning behind suggestions
- Distinguish between blocking issues and suggestions
- Acknowledge good practices and improvements

Document Version:	1.0
Last Updated:	27/06
Next Review:	30/06 (After Demo 2)