3. Develop Admin Service

1. User Management: Allow admins to create, delete, and manage user accounts.

2. Book Management: Provide interfaces for admins to manage the book catalog, including adding new books or removing old ones.

3. Loan Management: Allow admins to oversee all ongoing and past loans, and to intervene manually if needed.

4. Audit Trails: Keep records of all admin actions for auditing purposes.

User Story 1: Admin User Management

Description:
As an admin, I want the ability to manage user accounts, so that I can ensure proper system usage.

Acceptance Criteria:

1. Admin should be able to create new user accounts.
2. Admin should be able to delete existing user accounts.
3. Admin should be able to view a list of all user accounts.
4. Admin should be able to disable/enable user accounts.

Certainly! Below is a detailed design for implementing the "Admin User Management" user story:

Entity: AdminUser

```
@Entity
public class AdminUser {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String username;

    private String password;

    // additional fields like email, roles, etc.
```

```
    // Getters and setters
}
```

Entity: User

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String username;

    private Boolean isEnabled;

    // additional fields like email, roles, etc.

    // Getters and setters
}
```

DAO: AdminUserRepository

```
@Repository
public interface AdminUserRepository extends JpaRepository<AdminUser, Long> {
    // Custom queries for admin user management
}
```

DAO: UserRepository

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // Custom queries for user management
}
```

Service: AdminUserService

```
@Service
```

```
public class AdminUserService {

    @Autowired
    private UserRepository userRepository;

    public User createUser(User user) {
        // Create a new user
    }

    public void deleteUser(Long userId) {
        // Delete an existing user
    }

    public List<User> getAllUsers() {
        // Return all users
    }

    public User enableOrDisableUser(Long userId, Boolean status) {
        // Enable or disable a user account
    }
}
```

Controller: AdminUserController

```
@RestController
@RequestMapping("/api/admin/users")
public class AdminUserController {

    @Autowired
    private AdminUserService adminUserService;

    @PostMapping
    public User createUser(@RequestBody User user) {
        return adminUserService.createUser(user);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        adminUserService.deleteUser(id);
    }

    @GetMapping
    public List<User> getAllUsers() {
```

```
        return adminUserService.getAllUsers();
    }

    @PutMapping("/{id}/status")
    public User enableOrDisableUser(@PathVariable Long id, @RequestParam Boolean status)
{
        return adminUserService.enableOrDisableUser(id, status);
    }
}
```

Integration with Other Microservices

1. **Authentication**: Ensure that only authenticated admins can access these endpoints. You can use JWT for this.
2. **Audit Trail**: Integrate with the logging service to keep a track of all activities performed by the admin.

Other Considerations

1. **Validation**: Implement proper validation for user creation and update.
2. **Exception Handling**: Implement proper exception handling for all CRUD operations.

By doing this, you'll be meeting all the acceptance criteria laid out in the user story.

User Story 2: Admin Book Management

Description:
As an admin, I want to manage the book catalog so that it remains up-to-date and accurate.

Acceptance Criteria:

1. Admin should be able to add new books to the catalog.
2. Admin should be able to delete books from the catalog.
3. Admin should be able to update book details.
4. Admin should be able to view a list of all books in the catalog.

Below is a detailed design for implementing the "Admin Book Management" user story:

Entity: Book

```
@Entity
public class Book {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String title;
    private String author;
    private String genre;
    private String isbn;
    private Boolean isAvailable;

    // Getters and setters
}
```

DAO: BookRepository

```
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {
    // Custom queries for book management
}
```

Service: AdminBookService

```
@Service
public class AdminBookService {

    @Autowired
    private BookRepository bookRepository;

    public Book createBook(Book book) {
        return bookRepository.save(book);
    }

    public void deleteBook(Long bookId) {
        bookRepository.deleteById(bookId);
    }

    public Book updateBook(Long bookId, Book book) {
        // Implementation for updating book
        // Retrieve existing book, update fields, save back
    }

    public List<Book> getAllBooks() {
```

```
        return bookRepository.findAll();
    }
}
```

Controller: AdminBookController

```
@RestController
@RequestMapping("/api/admin/books")
public class AdminBookController {

    @Autowired
    private AdminBookService adminBookService;

    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return adminBookService.createBook(book);
    }

    @DeleteMapping("/{id}")
    public void deleteBook(@PathVariable Long id) {
        adminBookService.deleteBook(id);
    }

    @PutMapping("/{id}")
    public Book updateBook(@PathVariable Long id, @RequestBody Book book) {
        return adminBookService.updateBook(id, book);
    }

    @GetMapping
    public List<Book> getAllBooks() {
        return adminBookService.getAllBooks();
    }
}
```

Integration with Other Microservices

1. Authentication: Make sure only authenticated admins can access these endpoints. You can use JWT for this.
2. Audit Trail: Integrate with an audit logging service to keep track of all activities performed by the admin.

Other Considerations

1. **Validation**: Implement validation for book addition and update (e.g., ISBN should be unique).
2. **Exception Handling**: Properly handle exceptions, especially when a book is not found or an error occurs during CRUD operations.

By adhering to this design, you will meet all the acceptance criteria for the user story.

User Story: Admin Loan Management

Description:
As an admin, I want to oversee all ongoing and past loans, so that I can intervene manually if needed.

Acceptance Criteria:

1. Admin should be able to view a list of all ongoing loans.
2. Admin should be able to view a list of all past loans.
3. Admin should have the ability to manually update the status of a loan (like cancel, complete, etc.)
4. Admin should be able to view loan details, including any late fees.

Below is a detailed design for implementing the "Admin Loan Management" user story:

Entity: Loan

```
@Entity
public class Loan {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ManyToOne
    private User user;

    @ManyToOne
    private Book book;
```

```
    private LocalDate issueDate;
    private LocalDate dueDate;
    private LocalDate returnDate;
    private Status loanStatus;
    private Double lateFee;

    // Getters and setters
}
```

Where `Status` could be an enum:

```
public enum Status {
    ONGOING,
    COMPLETED,
    CANCELLED
}
```

DAO: LoanRepository

```
@Repository
public interface LoanRepository extends JpaRepository<Loan, Long> {
    List<Loan> findByLoanStatus(Status status);
}
```

Service: AdminLoanService

```
@Service
public class AdminLoanService {

    @Autowired
    private LoanRepository loanRepository;

    public List<Loan> getAllOngoingLoans() {
        return loanRepository.findByLoanStatus(Status.ONGOING);
    }

    public List<Loan> getAllPastLoans() {
```

```
        return loanRepository.findByLoanStatus(Status.COMPLETED);
    }

    public void updateLoanStatus(Long loanId, Status newStatus) {
        // Retrieve loan, update status, save back
    }

    public Loan getLoanDetails(Long loanId) {
        return loanRepository.findById(loanId).orElse(null);
    }
}
```

Controller: AdminLoanController

```
@RestController
@RequestMapping("/api/admin/loans")
public class AdminLoanController {

    @Autowired
    private AdminLoanService adminLoanService;

    @GetMapping("/ongoing")
    public List<Loan> getAllOngoingLoans() {
        return adminLoanService.getAllOngoingLoans();
    }

    @GetMapping("/past")
    public List<Loan> getAllPastLoans() {
        return adminLoanService.getAllPastLoans();
    }

    @PutMapping("/{id}/status")
    public void updateLoanStatus(@PathVariable Long id, @RequestBody Status newStatus) {
        adminLoanService.updateLoanStatus(id, newStatus);
    }

    @GetMapping("/{id}")
    public Loan getLoanDetails(@PathVariable Long id) {
        return adminLoanService.getLoanDetails(id);
    }
}
```

Integration with Other Microservices

1. **Authentication**: Ensure only authenticated admins can access these endpoints. This can be achieved through JWT.
2. **Audit Trail**: Integrate with an audit logging service to keep track of all activities performed by the admin.

Other Considerations

1. Validation: Implement validation for loan status updates.
2. **Exception Handling**: Properly handle exceptions, particularly when a loan is not found or an error occurs during CRUD operations.

By following this design, you'll satisfy all the acceptance criteria for this user story.

User Story 3: Admin Audit Trails

Description:
As an admin, I want to have an audit trail for all admin actions, to ensure transparency and accountability.

Acceptance Criteria:

1. All admin actions like creating, deleting, or updating user accounts should be logged.
2. All admin actions related to book and loan management should be logged.
3. The admin should be able to view the audit logs.
4. The audit logs should include a timestamp, admin ID, and the action taken.

Below is a detailed design for the "Admin Audit Trails" user story:

Entity: AuditLog

```
@Entity
public class AuditLog {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private Long adminId;
    private String action;
    private String entityType;
```

```
    private Long entityId;
    private LocalDateTime timestamp;

    // Getters and setters
}
```

DAO: AuditLogRepository

```
@Repository
public interface AuditLogRepository extends JpaRepository<AuditLog, Long> {
    List<AuditLog> findByAdminId(Long adminId);
}
```

Service: AuditLogService

```
@Service
public class AuditLogService {

    @Autowired
    private AuditLogRepository auditLogRepository;

    public void addLog(Long adminId, String action, String entityType, Long entityId) {
        AuditLog log = new AuditLog();
        log.setAdminId(adminId);
        log.setAction(action);
        log.setEntityType(entityType);
        log.setEntityId(entityId);
        log.setTimestamp(LocalDateTime.now());
        auditLogRepository.save(log);
    }

    public List<AuditLog> getAllLogsByAdmin(Long adminId) {
        return auditLogRepository.findByAdminId(adminId);
    }
}
```

Controller: AuditLogController

```
@RestController
@RequestMapping("/api/admin/audit-logs")
```

```
public class AuditLogController {

    @Autowired
    private AuditLogService auditLogService;

    @GetMapping("/{adminId}")
    public List<AuditLog> getAllLogsByAdmin(@PathVariable Long adminId) {
        return auditLogService.getAllLogsByAdmin(adminId);
    }
}
```

Integration with Existing Features

1. **Invoke addLog Method**: Whenever an admin action is performed (like creating, deleting, or updating users/books/loans), you can call `auditLogService.addLog(adminId, action, entityType, entityId)` to add an audit log.

2. **Authentication**: Use the JWT to identify the admin who is performing the actions.

Other Considerations

1. Validation: Implement validation to ensure that only valid actions and entity types can be logged.
2. Exception Handling: Properly handle exceptions for adding logs and fetching logs.

By following this design, you'll satisfy all the acceptance criteria mentioned in the user story.