Develop Loan Service

1. Issue Book: Implement a function to issue a book to a user. This function should verify that the book is available for loan and that the user has not exceeded a maximum loan limit.

2. Return Book: Implement a function to manage the return of a loaned book. This function should update the book's status and any loan records.

3. Due Dates and Penalties: Implement logic to manage due dates for returned books and calculate any late return penalties**.**

User Story 1: Issue Book to a User

Description:
As a user, I want to be able to issue a book so that I can read it.

Acceptance Criteria:

1. The book should exist in the catalog.
2. The book should be available for loan (not already issued to someone else).
3. The user should not have exceeded their maximum loan limit.
4. Once the book is issued, the book's status should be updated to 'Loaned'.
5. The user's list of loaned books should be updated.

Detailed Design

Entity

Let's create an entity to represent a loan record.

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

    private Long userId;
    private Long bookId;
    private LocalDate issueDate;
    private LocalDate dueDate;
    // getters and setters
}
```

Controller

In the LoanService, create a REST controller to issue a book.

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

    @Autowired
    private LoanService loanService;

    @PostMapping("/issue")
    public ResponseEntity<LoanRecord> issueBook(@RequestBody LoanRequest loanRequest)
{
        return ResponseEntity.ok(loanService.issueBook(loanRequest));
    }

    // ... other endpoints
}
```

Service

The service layer will contain the business logic.

```
@Service
public class LoanService {

    @Autowired
    private LoanRepository loanRepository;

    @Autowired
    private BookCatalogService bookCatalogService; // Assume this is a feign client to Book
Catalog MS

    @Autowired
    private UserService userService; // Assume this is a feign client to User MS
    public LoanRecord issueBook(LoanRequest loanRequest) {
        // Validation and business logic here
        // e.g., check if book is available, if user limit is not exceeded, etc.

        // Assuming the above checks pass
        LoanRecord loanRecord = new LoanRecord();
        // ... populate loanRecord fields
        return loanRepository.save(loanRecord);
    }

    // ... other methods
}
```

Repository (DAO)

We will need a repository to manage `LoanRecord` entities.

```
@Repository
public interface LoanRepository extends JpaRepository<LoanRecord, Long> {
    // custom queries if needed
}
```

Integration with Other Microservices

1. **User Microservice**: Use Feign client or RestTemplate to fetch the user's current loan limit and other details. Verify the user ID and loan limit before issuing a book.
2. **Book Catalog Microservice**: Similarly, verify if the book is available for loan.

Example Feign Client Configuration for `UserService`

```
@FeignClient(name = "userService", url = "http://user-service-url/")
public interface UserService {

    @GetMapping("/api/users/{userId}/loanLimit")
    Integer getUserLoanLimit(@PathVariable Long userId);
}
```

Other Considerations

1. **Exception Handling:** Make sure to handle scenarios where the book is not available, the user has reached their limit, etc.
2. **Transaction Management**: Use Spring's `@Transactional` to manage transactions where needed.

This should provide a fairly detailed design to help you implement the "Issue Book to a User" user story. Modify as needed based on requirements and constraints.

User Story 2: Return a Loaned Book

Description:
As a user, I want to be able to return a book that I have loaned so that others can use it.

Acceptance Criteria:

1. The book should be in the user's list of loaned books.
2. Once the book is returned, the book's status should be updated to 'Available'.
3. The user's list of loaned books should be updated to remove the returned book.
4. Any loan records should be updated accordingly (e.g., end date of the loan).

Let's look into the detailed design for the "Return a Loaned Book" user story.

Detailed Design

Entity

We've already created a `LoanRecord` entity for the previous user story, so we'll continue to use it here.

Controller

In the `LoanService`, create a REST controller to handle book returns.

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

    @Autowired
    private LoanService loanService;

    @PostMapping("/return")
    public ResponseEntity<LoanRecord> returnBook(@RequestBody ReturnRequest
returnRequest) {
        return ResponseEntity.ok(loanService.returnBook(returnRequest));
    }

    // ... other endpoints
}
```

Service

The service layer will contain the business logic for handling book returns.

```
@Service
public class LoanService {

    @Autowired
    private LoanRepository loanRepository;

    @Autowired
    private BookCatalogService bookCatalogService; // Assume this is a feign client to Book
Catalog MS

    public LoanRecord returnBook(ReturnRequest returnRequest) {
        // Validation and business logic here
        // e.g., check if book is currently loaned by the user, calculate any late fees, etc.

        // Assuming the above checks pass, update the LoanRecord and book status
        LoanRecord loanRecord =
loanRepository.findById(returnRequest.getLoanId()).orElseThrow(() -> new
ResourceNotFoundException("LoanRecord not found"));

        // ... populate loanRecord fields, e.g., endDate, status, etc.
        loanRecord.setEndDate(LocalDate.now());

        return loanRepository.save(loanRecord);
    }

    // ... other methods
}
```

Repository (DAO)

We'll continue using the same `LoanRepository`.

```
@Repository
public interface LoanRepository extends JpaRepository<LoanRecord, Long> {
    // custom queries if needed
}
```

Integration with Other Microservices

1. **User Microservice**: Verify that the book is actually loaned to the user making the return request. This could involve a check against the user ID and loaned book records.

2. **Book Catalog Microservice**: Once the book is returned, its status needs to be updated in the Book Catalog. A Feign client or RestTemplate can be used to make this call.

Example Feign Client Configuration for `BookCatalogService`

```
@FeignClient(name = "bookCatalogService", url = "http://book-catalog-service-url/")
public interface BookCatalogService {

    @PostMapping("/api/books/{bookId}/setStatus")
    void setBookStatus(@PathVariable Long bookId, @RequestParam String status);
}
```

Other Considerations

1. Exception Handling: Proper exception handling for scenarios like invalid book ID, the book not being loaned by the user, etc.

2. Transaction Management: If your service spans multiple operations that need to be transactional, you can use Spring's `@Transactional` annotation.

This design covers the main aspects needed for the implementation of the "Return a Loaned Book" user story. Modify as needed based on needs.

User Story 3: Manage Due Dates for Loaned Books

Description:
As a user, I want to know the due date of the book that I have loaned, so that I can avoid any penalties.

Acceptance Criteria:

1. Each loaned book should have a due date which is visible to the user.
2. The due date should be calculated based on a standard loan period (e.g., 14 days from the issue date).

let's discuss the detailed design for the "Manage Due Dates for Loaned Books" user story.

Detailed Design

Entity

We'll extend the `LoanRecord` entity to include the due date information.

```
@Entity
public class LoanRecord {
    // ... existing fields

    private LocalDate issueDate;
    private LocalDate dueDate;

    // Getters and setters
}
```

Controller

Add an endpoint to fetch loan records for a specific user in `LoanController`.

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

    @Autowired
    private LoanService loanService;

    @GetMapping("/user/{userId}")
    public ResponseEntity<List<LoanRecord>> getLoanRecordsForUser(@PathVariable Long
userId) {
        return ResponseEntity.ok(loanService.getLoanRecordsForUser(userId));
    }

    // ... other endpoints
}
```

Service

Extend `LoanService` to fetch and calculate due dates.

```
@Service
public class LoanService {

    @Autowired
    private LoanRepository loanRepository;

    public List<LoanRecord> getLoanRecordsForUser(Long userId) {
        List<LoanRecord> loanRecords = loanRepository.findByUserId(userId);

        // No additional logic needed here, as the due date is already part of LoanRecord

        return loanRecords;
    }

    public LoanRecord issueBook(IssueRequest issueRequest) {
        LoanRecord loanRecord = new LoanRecord();

        loanRecord.setIssueDate(LocalDate.now());
        loanRecord.setDueDate(LocalDate.now().plusDays(14)); // 14 days standard loan period

        // ... rest of the logic to issue a book

        return loanRepository.save(loanRecord);
    }

    // ... other methods
}
```

Repository (DAO)

Extend `LoanRepository` to fetch records by user ID.

```
@Repository
public interface LoanRepository extends JpaRepository<LoanRecord, Long> {
    List<LoanRecord> findByUserId(Long userId);
}
```

Integration with Other Microservices

1. **User Microservice**: You will likely need to fetch the user's loan records, either to display in the User Service or as part of the Loan Service.

2. **Notification Microservice**: Optionally, you can send a notification when a book is nearing its due date.

Example Feign Client Configuration for `NotificationService`

```
@FeignClient(name = "notificationService", url = "http://notification-service-url/")
public interface NotificationService {

    @PostMapping("/api/notifyDueDate")
    void notifyDueDate(@RequestBody DueDateNotificationRequest request);
}
```

Other Considerations

1. Exception Handling Properly handle edge cases, such as when the user doesn't have any loan records.

2. Transaction Management: Use Spring's `@Transactional` for any operations that should be atomic.

This should cover the essential components needed for the "Manage Due Dates for Loaned Books" user story. Modify the design as needed based on your specific requirements.

User Story 4: Calculate Penalties for Late Returns

Description:

As a librarian, I want the system to calculate any penalties for books that are returned late, so that the library's policies are consistently enforced.

Acceptance Criteria:

1. If a book is returned after the due date, a late fee should be calculated.
2. The late fee should be calculated based on the number of days the book is overdue.
3. The user should be informed of any late fees upon the return of a book.
4. The late fee should be stored in the user's account until it's cleared.

Detailed Design

Entity

Extend the `LoanRecord` and `UserAccount` entities to include penalty information.

```
@Entity
public class LoanRecord {
    // ... existing fields

    private BigDecimal lateFee;

    // Getters and setters
}

@Entity
public class UserAccount {
    // ... existing fields

    private BigDecimal totalLateFees;

    // Getters and setters
}
```

Controller

Extend `LoanController` to handle the return of books and calculation of late fees.

```
@RestController
@RequestMapping("/api/loans")
```

```
public class LoanController {

    @Autowired
    private LoanService loanService;

    @PostMapping("/return/{loanId}")
    public ResponseEntity<ReturnResponse> returnBook(@PathVariable Long loanId) {
        return ResponseEntity.ok(loanService.returnBook(loanId));
    }

    // ... other endpoints
}
```

Service

Extend `LoanService` to handle the return process and calculation of late fees.

```
@Service
public class LoanService {

    @Autowired
    private LoanRepository loanRepository;

    @Autowired
    private UserAccountRepository userAccountRepository;

    public ReturnResponse returnBook(Long loanId) {
        LoanRecord loanRecord = loanRepository.findById(loanId).orElseThrow(() -> new
LoanNotFoundException("Loan not found"));

        LocalDate dueDate = loanRecord.getDueDate();
        LocalDate today = LocalDate.now();

        BigDecimal lateFee = BigDecimal.ZERO;

        if (today.isAfter(dueDate)) {
            long daysLate = DAYS.between(dueDate, today);
            lateFee = new BigDecimal(daysLate).multiply(LATE_FEE_PER_DAY); // Assume
LATE_FEE_PER_DAY is a BigDecimal constant

            loanRecord.setLateFee(lateFee);
            loanRepository.save(loanRecord);
            UserAccount userAccount = userAccountRepository.findById(loanRecord.getUserId())
                .orElseThrow(() -> new UserNotFoundException("User not found"));
            userAccount.setTotalLateFees(userAccount.getTotalLateFees().add(lateFee));
            userAccountRepository.save(userAccount);
        }

        return new ReturnResponse(lateFee);
    }

    // ... other methods
}
```

Repository (DAO)

No new methods need to be added to the repositories, but make sure you already have methods to fetch a `LoanRecord` by its ID and a `UserAccount` by its ID.

Integration with Other Microservices

1. **User Microservice**: The late fees are added to the user account, which might be managed in the User Service.

2. **Notification Microservice**: Optionally, you can notify the user about late fees through this service.

Other Considerations

1. Exception Handling Ensure you handle exceptions like "Loan Not Found" or "User Not Found" appropriately.

2. Transaction Management: Use Spring's `@Transactional` for any operations that should be atomic.

3. Constants and Configuration: The late fee per day (`LATE_FEE_PER_DAY`) should ideally be configurable, either through an admin interface or a configuration file.

4. User Notifications: Depending on requirements, you might want to notify the user of the late fees via email or some other channel when they incur a late fee.

This should provide a comprehensive design for the "Calculate Penalties for Late Returns" user story. Modify the design as needed based on your requirements.