# **Breakout Sessions Handout**

## **Breakout Session 1:**

|  |
| --- |
| *# Breakout Room 1: Designing a Simple Database Schema*  # Student Worksheet  # =============================================================================  # SCENARIO: ONLINE BOOKSTORE DATABASE  # =============================================================================  # Your team needs to design a database schema for a new online bookstore.  # The bookstore needs to manage books, customers, orders, and reviews.  # Together, you'll identify entities, attributes, and relationships,  # then represent one entity as a Python class.  # =============================================================================  # TASK 1: ENTITY IDENTIFICATION  # =============================================================================  # List all the main entities (objects) that should be in your database.  # Add at least 4-5 entities below:  entities = [  # Example: "Book"  # Add more entities here...  ]  # =============================================================================  # TASK 2: ENTITY ATTRIBUTES  # =============================================================================  # For each entity, list the attributes (properties) they should have.  # Include data types and note which attribute would be the primary key.  # Example format:  book\_attributes = {  "book\_id": "integer (primary key)",  # Add more attributes here...  }  customer\_attributes = {  # Add attributes here...  }  # Add more entities and their attributes...  # =============================================================================  # TASK 3: RELATIONSHIPS  # =============================================================================  # Identify how the entities relate to each other.  # Describe each relationship and its type (One-to-One, One-to-Many, Many-to-Many)  relationships = [  # Example: "Book to Author: Many-to-Many (A book can have multiple authors, and an author can write multiple books)"  # Add more relationships here...  ]  # =============================================================================  # TASK 4: PYTHON REPRESENTATION  # =============================================================================  # Choose one entity and implement it as a Python class.  # Include attributes as instance variables and add appropriate methods.  # Example (incomplete - you need to finish it):  class Book:  def \_\_init\_\_(self, book\_id, title):  self.book\_id = book\_id  self.title = title  # Add more attributes here...  # Add methods here...  # For example: display\_info, update\_stock, add\_author, etc.  # =============================================================================  # TASK 5: DISCUSSION QUESTIONS  # =============================================================================  # Discuss these questions as a group and write your answers as comments:  # 1. What challenges did you encounter when designing this schema?  # Answer:  # 2. What trade-offs did you make in your design?  # Answer:  # 3. How might your schema need to evolve as the bookstore grows?  # Answer:  # 4. What potential data integrity issues should you watch out for?  # Answer:  # 5. How would queries for common information (like a customer's order  # history or all reviews for a book) work with your schema?  # Answer: |

## **Breakout Session 2:**

|  |
| --- |
| *# Breakout Room 2: Implementing Data Flows*  # INSTRUCTOR ANSWER SHEET  # =============================================================================  # SCENARIO: BOOK REVIEW SUBMISSION SYSTEM  # =============================================================================  # This is a sample solution for implementing a data flow for submitting  # book reviews in an online bookstore.  # =============================================================================  # TASK 1: DEFINE THE DATA MODEL (Sample Answer)  # =============================================================================  import datetime  class BookReview:  def \_\_init\_\_(self, review\_id=None, book\_id=None, user\_id=None, rating=None, comment=None,  timestamp=None, status=None):  self.review\_id = review\_id  self.book\_id = book\_id  self.user\_id = user\_id  self.rating = rating  self.comment = comment  self.timestamp = timestamp or datetime.datetime.now()  self.status = status or "pending"  def is\_valid(self):  """Check if the review contains required fields with valid values"""  if not self.book\_id or not isinstance(self.book\_id, int) or self.book\_id <= 0:  return False  if not self.user\_id or not isinstance(self.user\_id, int) or self.user\_id <= 0:  return False  if (not self.rating or not isinstance(self.rating, int) or  self.rating < 1 or self.rating > 5):  return False  return True  def display(self):  """Display the review information"""  print(f"Review ID: {self.review\_id}")  print(f"Book ID: {self.book\_id}")  print(f"User ID: {self.user\_id}")  print(f"Rating: {self.rating} stars")  if self.comment:  print(f"Comment: {self.comment}")  print(f"Timestamp: {self.timestamp}")  print(f"Status: {self.status}")  # =============================================================================  # TASK 2: IMPLEMENT INPUT VALIDATION (Sample Answer)  # =============================================================================  def validate\_review\_data(review\_data):  """  Validate the review data before processing  Return a list of error messages (empty if valid)  """  errors = []  # Check if book\_id is provided and valid  if "book\_id" not in review\_data:  errors.append("Book ID is required")  else:  try:  book\_id = int(review\_data["book\_id"])  if book\_id <= 0:  errors.append("Book ID must be a positive number")  except (ValueError, TypeError):  errors.append("Book ID must be a number")  # Check if user\_id is provided and valid  if "user\_id" not in review\_data:  errors.append("User ID is required")  else:  try:  user\_id = int(review\_data["user\_id"])  if user\_id <= 0:  errors.append("User ID must be a positive number")  except (ValueError, TypeError):  errors.append("User ID must be a number")  # Check if rating is provided and valid  if "rating" not in review\_data:  errors.append("Rating is required")  else:  try:  rating = int(review\_data["rating"])  if rating < 1 or rating > 5:  errors.append("Rating must be between 1 and 5")  except (ValueError, TypeError):  errors.append("Rating must be a number")  # Check if comment is valid (if provided)  if "comment" in review\_data and review\_data["comment"]:  comment = str(review\_data["comment"])  if len(comment) > 500:  errors.append("Comment must be less than 500 characters")  return errors  # =============================================================================  # TASK 3: IMPLEMENT DATA TRANSFORMATION (Sample Answer)  # =============================================================================  def transform\_review\_data(review\_data):  """  Transform and clean the review data  Return the transformed data  """  transformed\_data = {}  # Convert IDs to integers  if "book\_id" in review\_data:  transformed\_data["book\_id"] = int(review\_data["book\_id"])  if "user\_id" in review\_data:  transformed\_data["user\_id"] = int(review\_data["user\_id"])  if "rating" in review\_data:  transformed\_data["rating"] = int(review\_data["rating"])  # Trim whitespace from comment  if "comment" in review\_data and review\_data["comment"]:  transformed\_data["comment"] = str(review\_data["comment"]).strip()  # Add timestamp and status  transformed\_data["timestamp"] = datetime.datetime.now()  transformed\_data["status"] = "pending"  # Generate a review ID (in a real system, this would come from the database)  import random  transformed\_data["review\_id"] = random.randint(10000, 99999)  return transformed\_data  # =============================================================================  # TASK 4: IMPLEMENT ERROR HANDLING (Sample Answer)  # =============================================================================  def handle\_review\_error(error\_type, details):  """  Handle errors that occur during the review submission process  """  if error\_type == "validation":  # Return friendly error messages to the user  return {  "success": False,  "error\_type": "validation",  "message": "Please fix the following issues:",  "details": details  }  elif error\_type == "database":  # Log the error and return a generic message  print(f"DATABASE ERROR: {details}") # In a real system, use proper logging  return {  "success": False,  "error\_type": "system",  "message": "Unable to save your review. Please try again later.",  "reference\_id": generate\_error\_reference()  }  elif error\_type == "system":  # Log the error and alert an administrator  print(f"CRITICAL SYSTEM ERROR: {details}") # In a real system, use proper logging  alert\_administrator(details) # This would be implemented in a real system  return {  "success": False,  "error\_type": "system",  "message": "A system error occurred. Support has been notified.",  "reference\_id": generate\_error\_reference()  }  else:  # Unknown error type  print(f"UNKNOWN ERROR TYPE: {error\_type}, Details: {details}")  return {  "success": False,  "error\_type": "unknown",  "message": "An unexpected error occurred. Please try again."  }  def generate\_error\_reference():  """Generate a unique reference ID for error tracking"""  import random  import string  chars = string.ascii\_uppercase + string.digits  return ''.join(random.choice(chars) for \_ in range(8))  def alert\_administrator(details):  """Alert system administrator about a critical error"""  # In a real system, this might send an email, text message, or create a ticket  print(f"ALERT: Administrator notified about: {details}")  # =============================================================================  # TASK 5: IMPLEMENT THE FULL DATA FLOW (Sample Answer)  # =============================================================================  def submit\_review\_flow(review\_data):  """  Process the full flow of submitting a review  Return the result of the operation  """  print("\n--- Starting Review Submission Flow ---")  # Step 1: Validate the input data  print("Validating review data...")  validation\_errors = validate\_review\_data(review\_data)  # Step 2: If validation fails, handle the errors and return  if validation\_errors:  print("Validation failed with errors:")  for error in validation\_errors:  print(f"- {error}")  return handle\_review\_error("validation", validation\_errors)  try:  # Step 3: Transform the data  print("Transforming review data...")  transformed\_data = transform\_review\_data(review\_data)  # Step 4: Create a BookReview object  print("Creating review object...")  review = BookReview(  review\_id=transformed\_data["review\_id"],  book\_id=transformed\_data["book\_id"],  user\_id=transformed\_data["user\_id"],  rating=transformed\_data["rating"],  comment=transformed\_data.get("comment", ""),  timestamp=transformed\_data["timestamp"],  status=transformed\_data["status"]  )  # Step 5: Simulate storing the review  print("Storing review in database...")  # In a real system, this would save to a database  # For this example, we'll just display the review  review.display()  # Simulate database error (uncomment to test error handling)  # if review.rating == 2:  # raise Exception("Database connection failed")  # Step 6: Return a success message  return {  "success": True,  "review\_id": review.review\_id,  "message": "Thank you for your review! It has been submitted for approval."  }  except Exception as e:  # Handle any unexpected errors  return handle\_review\_error("system", str(e))  # =============================================================================  # TASK 6: TEST YOUR IMPLEMENTATION (Sample Answer)  # =============================================================================  # Example test data (valid)  valid\_review = {  "book\_id": 123,  "user\_id": 456,  "rating": 4,  "comment": "This book was very helpful and well-written."  }  # Example test data (invalid)  invalid\_review = {  "book\_id": "not\_a\_number",  "rating": 7, # Out of range  "comment": "Too short"  # Missing user\_id  }  # Test the implementation  print("\n=== Testing Valid Review ===")  result1 = submit\_review\_flow(valid\_review)  print("Result for valid review:", result1)  print("\n=== Testing Invalid Review ===")  result2 = submit\_review\_flow(invalid\_review)  print("Result for invalid review:", result2)  # =============================================================================  # DISCUSSION QUESTIONS (Sample Answers)  # =============================================================================  # 1. What are the benefits of breaking down the data flow into separate functions?  # Answer:  # - Improved maintainability: Each function has a single responsibility  # - Better testability: Functions can be tested independently  # - Code reuse: Functions can be used in different contexts  # - Easier debugging: Issues can be isolated to specific functions  # - Team collaboration: Different team members can work on different parts  # 2. How would you modify this flow to handle updates to existing reviews?  # Answer:  # - Add a function to retrieve existing reviews by ID  # - Add validation to check if the user is authorized to update the review  # - Modify the transform function to preserve original data like creation timestamp  # - Add version tracking or history to maintain a record of changes  # - Update the status field to "updated" instead of "pending"  # - Add checks to prevent updates to reviews that are too old  # 3. What additional validation rules might be important in a real system?  # Answer:  # - Check if the book actually exists in the database  # - Verify that the user has purchased or borrowed the book  # - Check for duplicate reviews (one review per user per book)  # - Validate for inappropriate content in comments  # - Rate limiting to prevent spam  # - Verify user authentication/authorization  # - Validate that the book has been released (no reviews for unreleased books)  # - Check for minimum comment length for detailed reviews  # 4. How would you implement a caching strategy for frequently accessed reviews?  # Answer:  # - Identify which reviews are accessed most frequently (e.g., for popular books)  # - Create a cache with an appropriate data structure (e.g., dictionary, LRU cache)  # - Store reviews in the cache after they're retrieved from the database  # - Set expiration times for cached items to ensure data freshness  # - Implement cache invalidation when reviews are updated or deleted  # - Use a tiered caching strategy (memory, distributed cache, database)  # - Cache aggregate data like average ratings separately from individual reviews  # - Monitor cache hit/miss rates to optimize caching strategy  # - Consider using an established caching system like Redis for production use  # 5. How would you handle a scenario where the database connection fails?  # Answer:  # - Implement retry logic with exponential backoff  # - Queue failed operations for later processing  # - Fall back to a secondary database if available  # - Return an appropriate error message to the user  # - Log detailed error information for debugging  # - Alert system administrators about persistent connection issues  # - Implement circuit breaker pattern to prevent cascading failures  # - Use a message queue to decouple submission from database operations  # - Have a degraded mode where basic functionality still works  # - Monitor and track failed operations to ensure they're eventually processed  # =============================================================================  # TEACHING NOTES  # =============================================================================  # Key concepts to emphasize:  # 1. The importance of data validation before processing  # 2. Separating concerns in data flow (validation, transformation, storage)  # 3. Proper error handling and user-friendly error messages  # 4. The difference between internal data models and external representations  # 5. Testing different scenarios (valid data, invalid data, error conditions)  #  # Watch for:  # - Students who miss critical validation steps  # - Confusion between validation and transformation steps  # - Error handling that exposes too much technical detail to users  # - Solutions that mix concerns (e.g., validation logic in the data model)  #  # Extension activities:  # - Have students implement an approval workflow for reviews  # - Ask students to add a feature to flag inappropriate reviews  # - Discuss how to implement review metrics (most helpful, etc.)  # - Explore how to handle high-volume review submission |