Documentación de Implementación - AmazinSpring

MIW - Arquitectura de Sitios Web - Trabajo 2

Índice

Introducción

Este documento describe la implementación completa de las funcionalidades requeridas para el Trabajo 2 de la asignatura Arquitectura de Sitios Web del Máster en Ingeniería Web. La implementación se ha realizado respetando la arquitectura en capas del piloto original (Presentación → Negocio → Persistencia) y manteniendo los convenios de nombres establecidos.

Acceso a la Aplicación

URL de Acceso

La aplicación está desplegada en una máquina virtual y es accesible a través de la siguiente URL:

🔗 **http://156.35.95.57:8080/Amazin_Spring_19_0/**

Información de Despliegue

- **Servidor**: Apache Tomcat
- **Puerto**: 8080
- **Contexto**: /Amazin_Spring_19_0
- **Base de Datos**: HSQLDB (hsql://localhost/amazin19)

Requisito 1: Compra de Libros

1.1 Control de Stock en Base de Datos

Modelo de Datos

Se ha modificado la entidad `Book` para incluir un atributo de stock:

```
@Entity
public class Book {
    @Id @GeneratedValue
    private int id;
    private String title;
    private String description;
    private String author;
    private double basePrice;
    private int stock;  // ← NUEVO ATRIBUTO
    // ...
}
```

Persistencia: El atributo `stock` se mapea automáticamente a la base de datos mediante JPA. La configuración de JPA está en `persistence.xml`:

```
<property name="jakarta.persistence.schema-generation.database.action"
          value="update" />
```

Esta propiedad permite que Hibernate actualice el esquema de la base de datos automáticamente al detectar el nuevo campo.

Inicialización: Según el requisito, cada libro debe tener 10 unidades disponibles al despliegue. Esto se gestiona a través de scripts de inicialización de base de datos o mediante la inserción manual de los datos con valores de stock = 10.

1.2 Carrito de la Compra

Modelo del Carrito

Se han creado dos clases principales para gestionar el carrito:

Cart.java - Contenedor principal del carrito:

```
public class Cart implements Serializable {
    private List<CartItem> items;

    // Añade un item al carrito, incrementa cantidad
    // si ya existe
    public void addItem(Book book, int quantity,
```

```
                        boolean isReserved);

    // Busca un item por ID de libro y tipo (reserva o compra)
    public CartItem findItemByBookId(int bookId,
                                     boolean isReserved);

    // Elimina un item específico
    public void removeItem(int bookId, boolean isReserved);

    // Calcula el total del carrito
    public double getTotal();

    // Vacía el carrito
    public void clear();
}
```

CartItem.java - Representa un item individual en el carrito:

```
public class CartItem implements Serializable {
    private Book book;
    private int quantity;
    private boolean isReserved;  // Reserva o compra normal

    // Calcula el subtotal considerando si es reserva
    // (95%) o compra (100%)
    public double getSubtotal() {
        if (isReserved) {
            // 95% restante a pagar
            return book.getPrice() * quantity * 0.95;
        }
        return book.getPrice() * quantity;
    }
}
```

Capa de Presentación

CartController.java - Gestiona las peticiones HTTP relacionadas con el carrito:

- **addToCart**: Añade un libro al carrito
    - Valida stock disponible antes de añadir
    - Utiliza la sesión HTTP para almacenar el carrito
    - Redirige al catálogo con mensaje de éxito/error

- **`viewCart`**: Muestra el contenido del carrito
    - Sincroniza las reservas desde la base de datos
    - Calcula el total incluyendo reservas pendientes
- **`removeFromCart`**: Elimina un item del carrito
    - Distingue entre items normales y reservas
    - Si es reserva, cancela en BD y restaura stock
- **`clearCart`**: Vacía todo el carrito
    - Cancela todas las reservas asociadas
    - Restaura el stock de los libros reservados
- **`checkout`**: Procesa la compra completa
    - Valida stock en tiempo real
    - Reduce stock en BD para compras normales
    - Procesa las reservas (elimina de BD sin restaurar stock)

Capa de Negocio

CartManager.java - Implementa la lógica de negocio del carrito:

```
@Override
public void addBookToCart(Cart cart, int bookId, int quantity)
        throws Exception {
    // 1. Obtener el libro con precio calculado
    Book book = bookManagerService.getBookById(bookId);

    // 2. Calcular cantidad total solicitada
    int totalRequested = quantity;
    CartItem existingItem =
        cart.findItemByBookId(bookId, false);
    if (existingItem != null) {
        totalRequested += existingItem.getQuantity();
    }

    // 3. Verificar stock disponible
    if (!bookManagerService.checkStockAvailability(
            bookId, totalRequested)) {
        throw new Exception("cart.notEnoughStock");
    }

    // 4. Añadir al carrito (no reduce stock hasta checkout)
```

```
        cart.addItem(book, quantity, false);
    }
```

Sincronización con Reservas:

```
    @Override
    public void synchronizeCartForUser(String username, Cart cart)
            throws Exception {
        // Obtener reservas del usuario desde BD
        List<Reservation> reservations =
            reservationManagerService.getReservations(username);

        // Eliminar todas las reservas del carrito
        cart.removeAllItems(true);

        // Agregar todas las reservas desde BD
        // (la BD es la fuente de verdad)
        for (Reservation res : reservations) {
            cart.addItem(res.getBook(),
                         res.getQuantity(), true);
        }
    }
```

Proceso de Checkout:

```
    @Override
    public boolean checkout(String username, Cart cart)
            throws Exception {
        // 1. Procesar reservas (eliminar de BD sin restaurar stock)
        reservationManagerService
            .processReservationsInCart(username, cart);

        // 2. Procesar compras normales
        for (CartItem item : cart.getItems()) {
            if (!item.isReserved()) {
                // Reducir stock con validación en tiempo real
                boolean reduced = bookManagerService
                    .reduceStock(item.getBookId(),
                                 item.getQuantity());
                if (!reduced) {
                    return false; // Stock insuficiente
                }
            }
```

```
    }

    return true;
}
```

1.3 Validación de Disponibilidad

Capa de Negocio

BookManager.java expone métodos para verificar y modificar stock:

```
@Override
public boolean checkStockAvailability(int bookId,
                                      int requestedQuantity)
        throws Exception {
    return bookDataService
        .checkStockAvailability(bookId, requestedQuantity);
}

@Override
public boolean reduceStock(int bookId, int quantity)
        throws Exception {
    return bookDataService.reduceStock(bookId, quantity);
}

@Override
public boolean increaseStock(int bookId, int quantity)
        throws Exception {
    bookDataService.increaseBookStock(bookId, quantity);
    return true;
}
```

Capa de Persistencia

BookDAO.java - Implementa las operaciones atómicas de stock:

Verificación de Stock:

```
@Override
public boolean checkStockAvailability(int bookId, int requestedQuantity)
        throws Exception {
    Dba dba = new Dba(true); // Solo lectura
```

```
    try {
        EntityManager em = dba.getActiveEm();
        Book book = em.find(Book.class, bookId);

        if (book != null) {
            boolean available =
                book.getStock() >= requestedQuantity;
            logger.debug("Stock check: Requested={}, " +
                        "Available={}", requestedQuantity,
                        book.getStock());
            return available;
        }
        return false;
    } finally {
        dba.closeEm();
    }
}
```

Reducción de Stock (con bloqueo pesimista):

```
@Override
public boolean reduceStock(int bookId, int quantity)
        throws Exception {
    Dba dba = new Dba();
    try {
        EntityManager em = dba.getActiveEm();

        // BLOQUEO PESIMISTA para evitar condiciones de carrera
        Book book = em.find(Book.class, bookId,
                            LockModeType.PESSIMISTIC_WRITE);

        if (book != null && book.getStock() >= quantity) {
            book.setStock(book.getStock() - quantity);
            em.merge(book);
            return true;
        }
        return false;
    } finally {
        dba.closeEm();
    }
}
```

Incremento de Stock (restauración):

```
@Override
public void increaseBookStock(int bookId, int quantity)
        throws Exception {
    Dba dba = new Dba();
    try {
        EntityManager em = dba.getActiveEm();

        // Bloqueo pesimista para operación atómica
        Book book = em.find(Book.class, bookId,
                           LockModeType.PESSIMISTIC_WRITE);

        if (book != null) {
            int newStock = book.getStock() + quantity;
            book.setStock(newStock);
            em.merge(book);
            logger.debug("Stock increased: New stock="
                        + newStock);
        }
    } finally {
        dba.closeEm();
    }
}
```

1.4 Actualización de Stock tras Checkout

La actualización del stock se realiza en dos momentos:

1. **Durante la reserva**: El stock se reduce inmediatamente al crear/actualizar una reserva
2. **Durante el checkout**: El stock se reduce para las compras normales

Proceso de Checkout completo:

```
@Override
public boolean checkout(String username, Cart cart)
        throws Exception {
    // Procesar reservas (ya redujeron stock al crearse)
    reservationManagerService
        .processReservationsInCart(username, cart);

    // Procesar compras normales (reducir stock ahora)
    for (CartItem item : cart.getItems()) {
        if (!item.isReserved()) {
            boolean reduced = bookManagerService
```

```
                .reduceStock(item.getBookId(),
                        item.getQuantity());
            if (!reduced) {
                return false;
            }
        }
    }
    return true;
}
```

Requisito 2: Sistema de Reservas

2.1 Modelo de Reservas

Entidad Reservation

Se ha creado una nueva entidad JPA para gestionar las reservas:

```
@Entity
public class Reservation {
    @Id @GeneratedValue
    private int id;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "book_id", nullable = false)
    private Book book;

    private String username;
    private int quantity;

    // Calcula el 5% pagado en la reserva
    public double getReservationPrice() {
        return book.getPrice() * quantity * 0.05;
    }

    // Calcula el 95% restante por pagar
    public double getRemainingAmount() {
        return book.getPrice() * quantity * 0.95;
    }
}
```

Relación con Book: Se establece una relación `@ManyToOne` con carga eager para que siempre se cargue el libro junto con la reserva, evitando problemas de lazy loading.

2.2 Funcionalidad de Reserva

Capa de Presentación

ReservationController.java - Gestiona las peticiones de reservas:

```
@RequestMapping("private/reserveBook")
public String reserveBook(
        @RequestParam("bookId") int bookId,
        @RequestParam("quantity") int quantity,
        Principal principal, HttpSession session) {

    synchronized (servletContext) {
        try {
            String username = principal.getName();

            // Toda la lógica encapsulada en el servicio
            reservationManagerService
                .reserveBook(username, bookId, quantity);

            session.setAttribute("message",
                                 "reservation.created");
            return "redirect:showBooks";

        } catch (Exception e) {
            handleReservationError(e, session, null,
                                   "Error reserving book", true);
            return "redirect:showBooks";
        }
    }
}
```

Capa de Negocio

ReservationManager.java - Lógica de negocio de reservas:

```
@Override
public Reservation reserveBook(String username, int bookId,
                               int quantity) throws Exception {
    // 1. Verificar si ya existe una reserva del
    // usuario para este libro
    Reservation existingReservation =
```

```
        getReservationByUserAndBook(username, bookId);
    boolean isNewReservation =
        (existingReservation == null);

    // 2. Si es nueva, validar que el libro existe
    if (isNewReservation) {
        Book book = bookManagerService.getBookById(bookId);
        if (book == null) {
            throw new Exception("reservation.bookNotFound");
        }
    }

    // 3. Verificar y reducir stock
    // (IMPORTANTE: reduce inmediatamente)
    if (!bookManagerService
            .checkStockAvailability(bookId, quantity)) {
        throw new Exception("reservation.notEnoughStock");
    }

    boolean stockReduced = bookManagerService
        .reduceStock(bookId, quantity);
    if (!stockReduced) {
        throw new Exception("reservation.stockReductionFailed");
    }

    try {
        Reservation reservation;

        if (isNewReservation) {
            // Crear nueva reserva
            reservation = reservationDataService
                .createReservation(username, bookId, quantity);
        } else {
            // Actualizar reserva existente
            // (incrementar cantidad)
            int newQuantity =
                existingReservation.getQuantity() + quantity;
            existingReservation.setQuantity(newQuantity);
            reservation = reservationDataService
                .updateReservation(existingReservation);
        }

        return reservation;
```

```
    } catch (Exception e) {
        // Rollback: restaurar stock si falla
        // la creación/actualización
        bookManagerService.increaseStock(bookId, quantity);
        throw e;
    }
}
```

Características importantes:

- **Reducción inmediata de stock**: Al crear una reserva, el stock se reduce inmediatamente para que no esté disponible para otros usuarios
- **Reservas acumulativas**: Si un usuario reserva el mismo libro varias veces, se incrementa la cantidad en lugar de crear múltiples reservas
- **Rollback automático**: Si falla la creación de la reserva en BD, se restaura el stock automáticamente

---

2.3 Integración con el Carrito

Marca Especial en el Carrito

Los items reservados se distinguen en el carrito mediante el atributo `isReserved`:

```
public class CartItem {
    private boolean isReserved;

    // El subtotal se calcula según el tipo
    public double getSubtotal() {
        if (isReserved) {
            // 95% pendiente
            return book.getPrice() * quantity * 0.95;
        }
        // 100% precio completo
        return book.getPrice() * quantity;
    }
}
```

Vista del Carrito (viewCart.jsp)

```
<td>
    <c:choose>
        <c:when test="${item.reserved}">
```

```
            <span style="color: orange; font-weight: bold;">
                <spring:message code="cart.reservation"/>
                (95% <spring:message code="cart.pending"/>)
            </span>
        </c:when>
        <c:otherwise>
            <spring:message code="cart.purchase"/>
        </c:otherwise>
    </c:choose>
</td>
```

Sincronización Automática

Cada vez que un usuario accede a `viewCart`, el sistema sincroniza automáticamente las reservas desde la base de datos:

```
@RequestMapping("private/viewCart")
public String viewCart(Principal principal,
                       HttpSession session,
                       Model model) {
    String username = principal.getName();
    Cart cart = getOrCreateCart(session);

    // Sincronizar con reservas (la BD es la fuente de verdad)
    cartManagerService.synchronizeCartForUser(username, cart);

    session.setAttribute(CART_ATTRIBUTE, cart);
    model.addAttribute("total", cart.getTotal());
    return "private/viewCart";
}
```

2.4 Reducción de Stock en Reservas

Momento de reducción: El stock se reduce **inmediatamente** al crear o incrementar una reserva, no al finalizar la compra. Esto garantiza que los libros reservados no estén disponibles para otros usuarios.

Implementación en ReservationManager:

```
// Verificar y reducir stock (común para ambos casos)
if (!bookManagerService.checkStockAvailability(bookId, quantity)) {
    throw new Exception("reservation.notEnoughStock");
```

```
}

boolean stockReduced = bookManagerService.reduceStock(bookId, quantity);
if (!stockReduced) {
    throw new Exception("reservation.stockReductionFailed");
}
```

Consecuencias:

- Los libros reservados se muestran con stock reducido en el catálogo
- Otros usuarios no pueden comprar ni reservar libros que ya están reservados
- Si se cancela una reserva, el stock se restaura inmediatamente

---

2.5 Sección "Mis Reservas"

Controlador

```
@RequestMapping("private/myReservations")
public String myReservations(Principal principal, Model model) {
    try {
        String username = principal.getName();
        List<Reservation> reservations =
            reservationManagerService.getReservations(username);

        model.addAttribute("reservations", reservations);
        return "private/myReservations";

    } catch (Exception e) {
        logger.error("Error getting reservations", e);
        model.addAttribute("error", "error.general");
        return "private/error";
    }
}
```

Vista (myReservations.jsp)

La vista muestra una tabla con:

- Título del libro
- Autor
- Precio unitario
- Cantidad reservada
- Importe pagado (5%)

- Importe pendiente (95%)
- Botones "Comprar" y "Eliminar"

```
<c:forEach var='reservation' items="${reservations}">
    <tr>
        <td><c:out value="${reservation.book.title}" /></td>
        <td><c:out value="${reservation.book.author}" /></td>
        <td><c:out value="${reservation.book.price}" /> €</td>
        <td><c:out value="${reservation.quantity}" /></td>
        <td>
            <c:out value="${reservation.reservationPrice}" /> €
        </td>
        <td>
            <c:out value="${reservation.remainingAmount}" /> €
        </td>
        <td>
            <form action="purchaseReservation" method="post">
                <input type="hidden" name="reservationId"
                       value="${reservation.id}"/>
                <input type="submit" value="Comprar"/>
            </form>
            <form action="cancelReservationFromPage"
                  method="post">
                <input type="hidden" name="reservationId"
                       value="${reservation.id}"/>
                <input type="submit" value="Eliminar"/>
            </form>
        </td>
    </tr>
</c:forEach>
```

2.6 Compra de Reserva (Pago del 95%)

Controlador

```
@RequestMapping("private/purchaseReservation")
public String purchaseReservation(
        @RequestParam("reservationId") int reservationId,
        Principal principal, Model model) {
    synchronized (servletContext) {
        try {
            String username = principal.getName();
```

```
            // El servicio valida internamente la propiedad
            reservationManagerService.purchaseReservation(
                reservationId, username);

            model.addAttribute("message",
                               "reservation.purchased");
            return "redirect:myReservations";

        } catch (Exception e) {
            handleReservationError(e, null, model,
                    "Error purchasing reservation", false);
            return "redirect:myReservations";
        }
    }
}
```

Lógica de Negocio

```
@Override
public boolean purchaseReservation(int reservationId, String username)
        throws Exception {
    // Obtener y validar propiedad
    Reservation reservation =
        getReservationById(reservationId, username);

    // Procesar compra: eliminar reserva de BD
    // El stock NO se restaura porque ya se había reducido
    // al crear la reserva
    reservationDataService
        .deleteReservation(reservationId);

    logger.debug("Reservation purchased. " +
                "Stock remains reduced.");
    return true;
}
```

Validación de propiedad:

```
private Reservation getReservationById(int reservationId, String username)
        throws Exception {
    Reservation reservation =
```

```
        reservationDataService.getReservationById(reservationId);

    if (reservation == null) {
        throw new Exception("reservation.notFound");
    }

    if (!reservation.getUsername().equals(username)) {
        logger.warn("User {} tried to access reservation {} " +
                    "belonging to {}", username, reservationId,
                    reservation.getUsername());
        throw new Exception("reservation.accessDenied");
    }

    return reservation;
}
```

2.7 Eliminación de Reserva (Restauración de Stock)

Controlador

```
@RequestMapping("private/cancelReservationFromPage")
public String cancelReservationFromPage(
        @RequestParam("reservationId") int reservationId,
        Principal principal, Model model) {
    synchronized (servletContext) {
        try {
            String username = principal.getName();

            // El servicio valida internamente la propiedad
            reservationManagerService.cancelReservation(
                reservationId, username);

            model.addAttribute("message",
                               "reservation.cancelled");
            return "redirect:myReservations";

        } catch (Exception e) {
            handleReservationError(e, null, model,
                    "Error cancelling reservation", false);
            return "redirect:myReservations";
        }
```

```
    }
}
```

Lógica de Negocio

```
@Override
public boolean cancelReservation(int reservationId, String username)
        throws Exception {
    // Obtener y validar propiedad
    Reservation reservation =
        getReservationById(reservationId, username);

    // Restaurar stock
    // (IMPORTANTE: devuelve las unidades al stock)
    Book book = reservation.getBook();
    bookManagerService.increaseStock(
        book.getId(), reservation.getQuantity());

    // Eliminar reserva
    reservationDataService.deleteReservation(reservationId);

    logger.debug("Reservation cancelled and stock restored");
    return true;
}
```

Flujo completo:

1. Se valida que la reserva pertenece al usuario autenticado
2. Se incrementa el stock del libro en la cantidad reservada
3. Se elimina la reserva de la base de datos
4. Las unidades vuelven a estar disponibles para compra/reserva

---

2.8 Capa de Persistencia - ReservationDAO

```
@Override
public Reservation createReservation(String username,
        int bookId, int quantity) throws Exception {
    Dba dba = new Dba();
    try {
        EntityManager em = dba.getActiveEm();

        // Obtener el Book en el MISMO contexto
```

```
            // de persistencia
            Book book = em.find(Book.class, bookId);
            if (book == null) {
                throw new Exception("Book not found: "
                                    + bookId);
            }

            // Crear la reserva
            Reservation reservation = new Reservation();
            reservation.setBook(book);
            reservation.setUsername(username);
            reservation.setQuantity(quantity);

            em.persist(reservation);
            return reservation;
        } finally {
            dba.closeEm();
        }
    }

    @Override
    public List<Reservation> getReservationsByUsername(String username)
            throws Exception {
        Dba dba = new Dba(true); // Solo lectura
        try {
            EntityManager em = dba.getActiveEm();

            TypedQuery<Reservation> query = em.createQuery(
                "SELECT r FROM Reservation r " +
                "WHERE r.username = :username " +
                "ORDER BY r.id DESC",
                Reservation.class);
            query.setParameter("username", username);

            List<Reservation> results = query.getResultList();

            // Forzar carga de libros para evitar
            // lazy loading issues
            for (Reservation r : results) {
                r.getBook().getTitle();
            }

            return results;
        } finally {
```

```
            dba.closeEm();
        }
    }

    @Override
    public Reservation updateReservation(Reservation reservation)
            throws Exception {
        Dba dba = new Dba();
        try {
            EntityManager em = dba.getActiveEm();

            // Obtener la reserva gestionada por
            // este EntityManager
            Reservation managed = em.find(Reservation.class,
                                          reservation.getId());
            if (managed == null) {
                throw new Exception("Reservation not found");
            }

            // Actualizar solo el campo quantity
            managed.setQuantity(reservation.getQuantity());

            return managed;
        } finally {
            dba.closeEm();
        }
    }

    @Override
    public void deleteReservation(int id) throws Exception {
        Dba dba = new Dba();
        try {
            EntityManager em = dba.getActiveEm();
            Reservation reservation = em.find(Reservation.class, id);
            if (reservation != null) {
                em.remove(reservation);
            }
        } finally {
            dba.closeEm();
        }
    }
```

Requisito 3: Internacionalización

3.1 Configuración de Spring

Definición del LocaleResolver

En `servlet-context.xml`:

```
<!-- Configuración de LocaleResolver para idiomas -->
<bean id="localeResolver"
  class=
    "org.springframework.web.servlet.i18n.SessionLocaleResolver">
    <property name="defaultLocale" value="es" />
</bean>
```

SessionLocaleResolver: Almacena el idioma seleccionado en la sesión HTTP del usuario, manteniendo la preferencia durante toda la sesión.

Definición del MessageSource

En `beans.xml`:

```
<bean id="messageSource"
  class=
   "org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="messages" />
    <property name="defaultEncoding" value="UTF-8" />
    <property name="fallbackToSystemLocale" value="false" />
</bean>
```

Propiedades:

- `basename`: Nombre base de los archivos de propiedades (messages.properties, messages_en.properties)
- `defaultEncoding`: UTF-8 para soportar caracteres especiales en español
- `fallbackToSystemLocale`: false para usar siempre el idioma configurado, no el del sistema

---

3.2 Archivos de Recursos

messages.properties (Español - por defecto)

```
welcome=Bienvenido a <em>la más pequeña</em> tienda virtual del mundo!
login.title=Introduzca usuario y contraseña
book.title=Título
```

```
cart.addToCart=Añadir al carrito
cart.viewCart=Ver carrito
cart.shoppingCart=Carrito de compra
reservation.reserve=Reservar
reservation.myReservations=Mis Reservas
language.select=Idioma
language.spanish=Español
language.english=Inglés
# ... más de 100 claves definidas
```

messages_en.properties (Inglés)

```
welcome=Welcome to the <em>smallest</em> virtual shop in the world!!!
login.title=Introduce login and password
book.title=Title
cart.addToCart=Add to Cart
cart.viewCart=View Cart
cart.shoppingCart=Shopping Cart
reservation.reserve=Reserve
reservation.myReservations=My Reservations
language.select=Language
language.spanish=Spanish
language.english=English
# ... más de 100 claves definidas
```

Cobertura completa: Se han internacionalizado todos los textos de la aplicación:

- Formularios de login
- Catálogo de libros
- Carrito de compra
- Sistema de reservas
- Mensajes de error y éxito
- Navegación y menús
- Footer y headers

3.3 Selector de Idioma

Componente Reutilizable (languageSelector.jsp)

```
<script type="text/javascript">
function changeLanguage(lang) {
    var contextPath = '${pageContext.request.contextPath}';
```

```
    window.location.href = contextPath + '/changeLanguage?lang=' + lang;
}
</script>

<div style="text-align: right; padding: 10px; margin-bottom: 10px;">
    <label for="languageSelect">
        <spring:message code="language.select"/>:
    </label>
    <select id="languageSelect" onchange="changeLanguage(this.value)"
            style="padding: 5px; font-size: 14px;">
        <option value="es"
                ${pageContext.response.locale.language == 'es'
                  ? 'selected' : ''}>
            Español
        </option>
        <option value="en"
                ${pageContext.response.locale.language == 'en'
                  ? 'selected' : ''}>
            English
        </option>
    </select>
</div>
```

Características:

- **Sin botones**: El cambio se realiza automáticamente al seleccionar un valor mediante el evento onchange
- **Selección persistente**: El selector muestra el idioma actualmente activo
- **Componente reutilizable**: Se incluye en todas las páginas mediante `<jsp:include>`

3.4 Controlador de Cambio de Idioma

LanguageController.java:

```
@Controller
public class LanguageController {

    @Autowired
    private LocaleResolver localeResolver;

    @GetMapping("/changeLanguage")
    public String changeLanguage(
            @RequestParam("lang") String lang,
            HttpServletRequest request,
```

```
            HttpServletResponse response) {

        // Establecer el nuevo locale en la sesión
        Locale locale = new Locale(lang);
        localeResolver.setLocale(request, response, locale);

        // Redirigir a la página anterior (Referer)
        String referer = request.getHeader("Referer");
        if (referer != null && !referer.isEmpty()) {
            return "redirect:" + referer;
        }

        // Si no hay referer, redirigir al menú principal
        return "redirect:/private/menu";
    }
}
```

Flujo de funcionamiento:

1. El usuario selecciona un idioma en el desplegable
2. JavaScript captura el evento `onchange` y hace una petición GET a `/changeLanguage?lang=XX`
3. El controlador establece el nuevo locale en la sesión
4. Se redirige al usuario a la misma página donde estaba (usando el header Referer)
5. La página se recarga mostrando todos los textos en el nuevo idioma

---

3.5 Configuración de Seguridad

En `SecurityConfig.java` se permite el acceso sin autenticación al endpoint de cambio de idioma:

```
.requestMatchers("/resources/**", "/changeLanguage").permitAll()
```

Esto permite que los usuarios cambien el idioma incluso en la página de login.

3.6 Uso en las Vistas JSP

Importación de librerías:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
```

Uso de mensajes internacionalizados:

```
<!-- Texto simple -->
<h2><spring:message code="cart.shoppingCart"/></h2>

<!-- En atributos de formularios -->
<input type="submit" value="<spring:message code='cart.addToCart'/>" />

<!-- En tablas -->
<th><spring:message code="cart.title"/></th>
<th><spring:message code="cart.quantity"/></th>

<!-- Mensajes dinámicos desde el modelo -->
<spring:message code="${sessionScope.message}"/>
```

Inclusión del selector:

```
<!-- En todas las páginas -->
<jsp:include page="../languageSelector.jsp" />
```

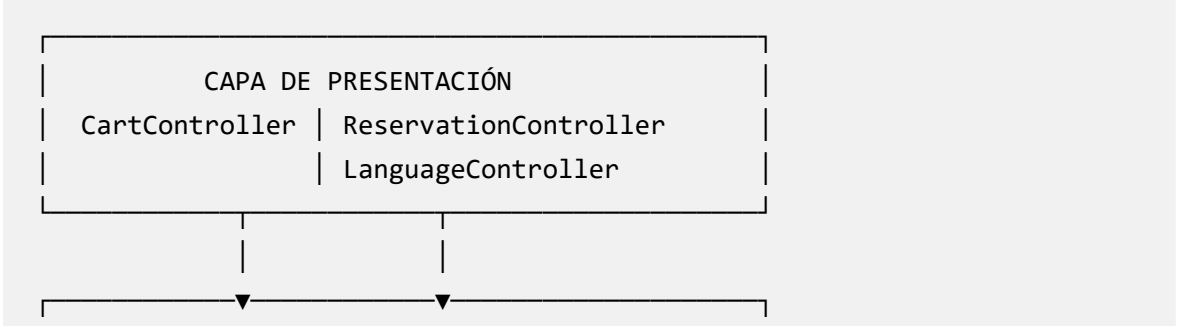
Arquitectura General

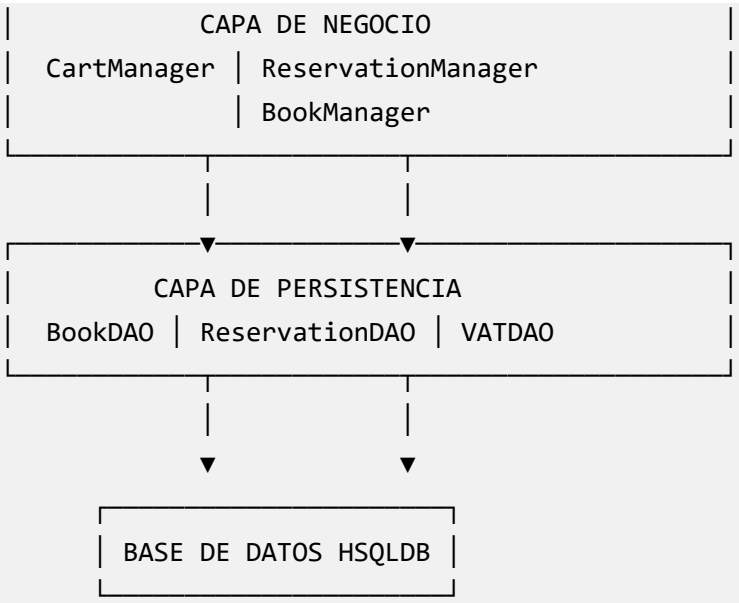
Patrón de Capas

La aplicación sigue una arquitectura en 3 capas bien definida:

```
PRESENTACIÓN (Controllers)
    ↓
NEGOCIO (Managers/Services)
    ↓
PERSISTENCIA (DAOs)
    ↓
BASE DE DATOS
```

Diagrama de Componentes





Responsabilidades por Capa

Capa de Presentación

- **Responsabilidad**: Gestionar peticiones HTTP, validación básica, control de sesión
- **Componentes**: Controllers (CartController, ReservationController, LanguageController)
- **No debe**: Contener lógica de negocio, acceder directamente a la BD

Capa de Negocio

- **Responsabilidad**: Lógica de negocio, validaciones complejas, coordinación entre DAOs
- **Componentes**: Managers/Services (CartManager, ReservationManager, BookManager)
- **No debe**: Gestionar peticiones HTTP, conocer detalles de implementación de persistencia

Capa de Persistencia

- **Responsabilidad**: Acceso a datos, operaciones CRUD, gestión de transacciones
- **Componentes**: DAOs (BookDAO, ReservationDAO, VATDAO)
- **No debe**: Contener lógica de negocio, gestionar sesiones HTTP

Inyección de Dependencias

Todas las dependencias se gestionan mediante **Spring IoC** con configuración XML:

```xml
<!-- beans.xml -->
<bean id="cartManagerService"
      class="com.miw.business.cartmanager.CartManager"/>
```

```
<bean id="reservationManagerService"
      class=
      "com.miw.business.reservationmanager.ReservationManager"/>
<bean id="bookDataService"
      class="com.miw.persistence.book.BookDAO"/>
<bean id="reservationDataService"
      class=
      "com.miw.persistence.reservation.ReservationDAO"/>
```

Inyección en Controladores:

```
@Controller
public class CartController {
    @Autowired
    private CartManagerService cartManagerService;
}
```

Inyección en Managers:

```
public class CartManager implements CartManagerService {
    @Autowired
    private BookManagerService bookManagerService;

    @Autowired
    private ReservationManagerService reservationManagerService;
}
```

Conclusiones

Funcionalidades Implementadas

✅ **Requisito 1: Compra de Libros**

- Control de stock en base de datos
- Carrito de compra funcional
- Validación de disponibilidad en tiempo real
- Actualización de stock tras checkout
- Al menos 3 libros con 10 unidades iniciales cada uno

✅ **Requisito 2: Sistema de Reservas**

- Funcionalidad de reservar libros
- Marca especial en carrito (5% pagado, 95% pendiente)

- Reducción de stock inmediata al reservar
- Sección "Mis Reservas" con opciones Comprar/Eliminar
- Pago del 95% restante al comprar
- Restauración de stock al eliminar reserva

✅ **Requisito 3: Internacionalización**

- Soporte completo para español (es) e inglés (en)
- Selector de idioma en todas las páginas
- Cambio automático sin botones (evento onchange)
- Más de 100 claves de traducción
- Persistencia del idioma en sesión

Aspectos Técnicos Destacables

- **Bloqueos pesimistas** en operaciones de stock para garantizar consistencia
- **Sincronización con BD** en cada acceso al carrito
- **Rollback automático** si falla la creación de reservas
- **Validación de propiedad** en operaciones de reservas
- **Mensajes internacionalizados** incluso en excepciones
- **Protección CSRF** en todos los formularios

Documento generado el: 27 de octubre de 2025
Autor: Implementación del Trabajo 2 - MIW
Versión: 1.1
URL de Despliegue: http://156.35.95.57:8080/Amazin_Spring_19_0/