Object Oriented Design

Design

Dr. Seán Russell

School of Computer Science,
University College Dublin

Lecture 07

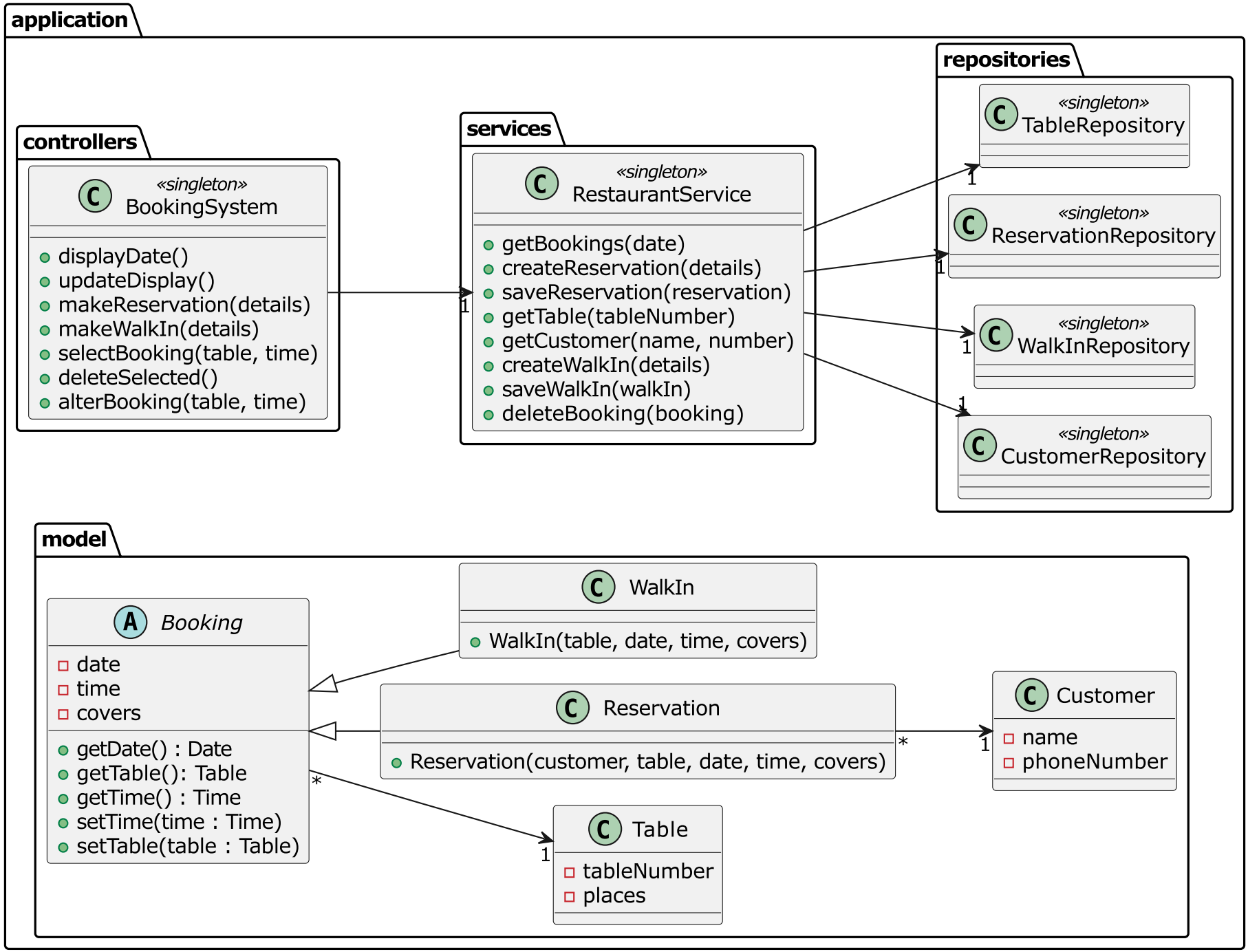
Table of Contents

- The primary task of design is to extend our modelling to the non application classes
  - Input, output and persistent storage
- Additionally, we must include more detail like the type and parameter information
- At this point, we must make decisions in order to progress further
- We need to know what frameworks we will be integrating with our code

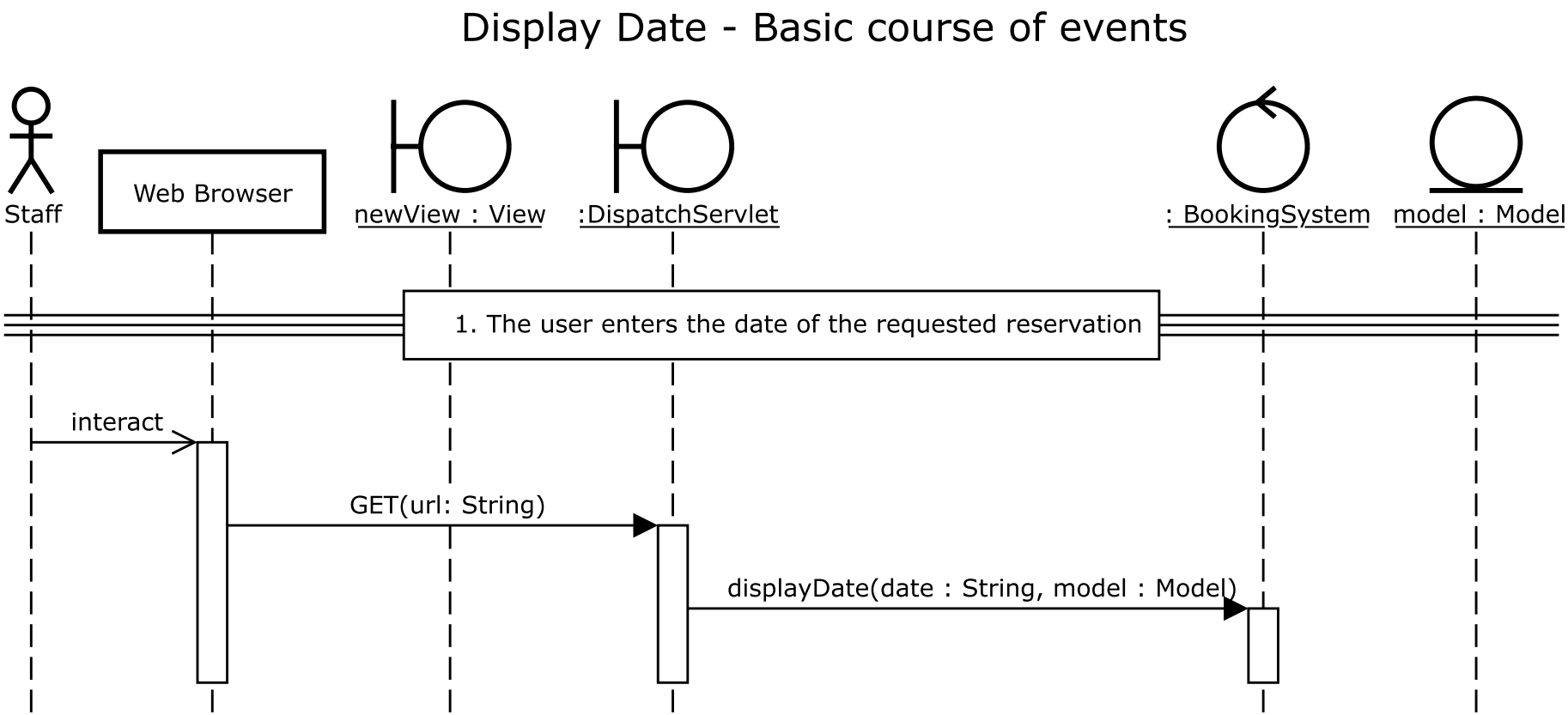
- Our design will be based on the use of the spring-boot framework
- This means that the user interface will be composed of HTML, CSS and Javascript
  - We will need to integrate our code with this
- There are many options for persistent storage of data that can be used with spring-boot
- We will assume the use of a relational database management system
  - We also need to integrate our code with this

Section Contents

- We need to re-evaluate the responsibilities of the classes
- RestaurantService had responsibilities for remembering and managing all entities
  - This is now managed by JPA interfaces that we have defined
- Previously, we defined the BookingSystem to manage the completion of the use cases
  - We can put the management responsibilities into the `RestaurantService`
  - We can use `BookingSystem` as our view controller


application
controllers
«singleton»
BookingSystem
displayDate()
updateDisplay()
makeReservation(details)
makeWalkIn(details)
selectBooking(table, time)
deleteSelected()
alterBooking(table, time)
services
«singleton»
RestaurantService
getBookings(date)
createReservation(details)
saveReservation(reservation)
getTable(tableNumber)
getCustomer(name, number)
createWalkIn(details)
saveWalkIn(walkIn)
deleteBooking(booking)
repositories
«singleton»
TableRepository
«singleton»
ReservationRepository
«singleton»
WalkInRepository
«singleton»
CustomerRepository
model
Booking
date
time
covers
getDate() : Date
getTable(): Table
getTime() : Time
setTime(time : Time)
setTable(table : Table)
WalkIn
WalkIn(table, date, time, covers)
Reservation
Reservation(customer, table, date, time, covers)
Customer
name
phoneNumber
Table
tableNumber
places

- All interactions in the system can be assumed to be between the web browser and the application
- When creating a page or showing interactions on the page we will represent this as a `View` object
  - This is the Java representation of the HTML, CSS, and Javascript returned by the ViewRenderer
- As much as possible we will not include the execution of Javascript in the interactions
- Sequence diagrams should begin as a HTTP GET or POST request to the application
  - This may be caused by a user action or by the execution of some Javascript code


Display Date - Basic course of events
Staff
Web Browser
newView : View
:DispatchServlet
: BookingSystem
model : Model
1. The user enters the date of the requested reservation
interact
GET(url: String)
displayDate(date : String, model : Model)

Section Contents

- The actual user interface presented to our user will be HTML, Javascript and CSS that is rendered in the browser
- However, this must be generated by our application
  - We will be using Thymeleaf to do this
- We still need to write the HTML and Javascript, but Thymeleaf can fill in data

- To generate a web page, we must first have a HTML template
  - This will contain special annotations to indicate where values should be filled in
  - These may be simple insertion or something more complex like a for each loop
- To generate the template, we must supply the expected information
- This is done by adding it to a Model object (passed to us from the DispatchServlet)

```
1  const tableList = [[${tables}]]
2
3  tab.setAttribute('value', /*[[£{error?.tableNumber}]]*/ '1');
```

```
1  <input type="date" name="date" id="start" th:value="${date}" onchange=
   ↪  "handleDateChange(event)">
2
3  <option th:each="table : ${tables}" th:value="${table.id}" th:text=
   ↪  "'Table ' + ${table.number} + ' (' + ${table.places} + ')'"
   ↪  th:selected="(${error?.tableId} == ${table.id})" ></option>
```

- If we use a value in the template, then it must be supplied in the model
- This task is completed by the BookingSystem object
- We use the method `addAttribute`
  - E.g. `model.addAttribute("date", date.toString());`
- If the attribute is an object, it is converted into a JSON representation

Section Contents

- At this point, we need to consider what data needs to be remembered
- In essence, we will choose the objects and which attributes they contain should be persisted
- In reality, we only have 4 classes that need to be remembered
  - Reservation
  - WalkIn
  - Table
  - Customer

- Table
  - Table number
  - Places
- Customer
  - Name
  - Phone number
- WalkIn
  - Date
  - Time
  - Covers
  - Table
  - Overfull

- Reservation
  - Date
  - Time
  - Covers
  - Table
  - Customer
  - Overfull

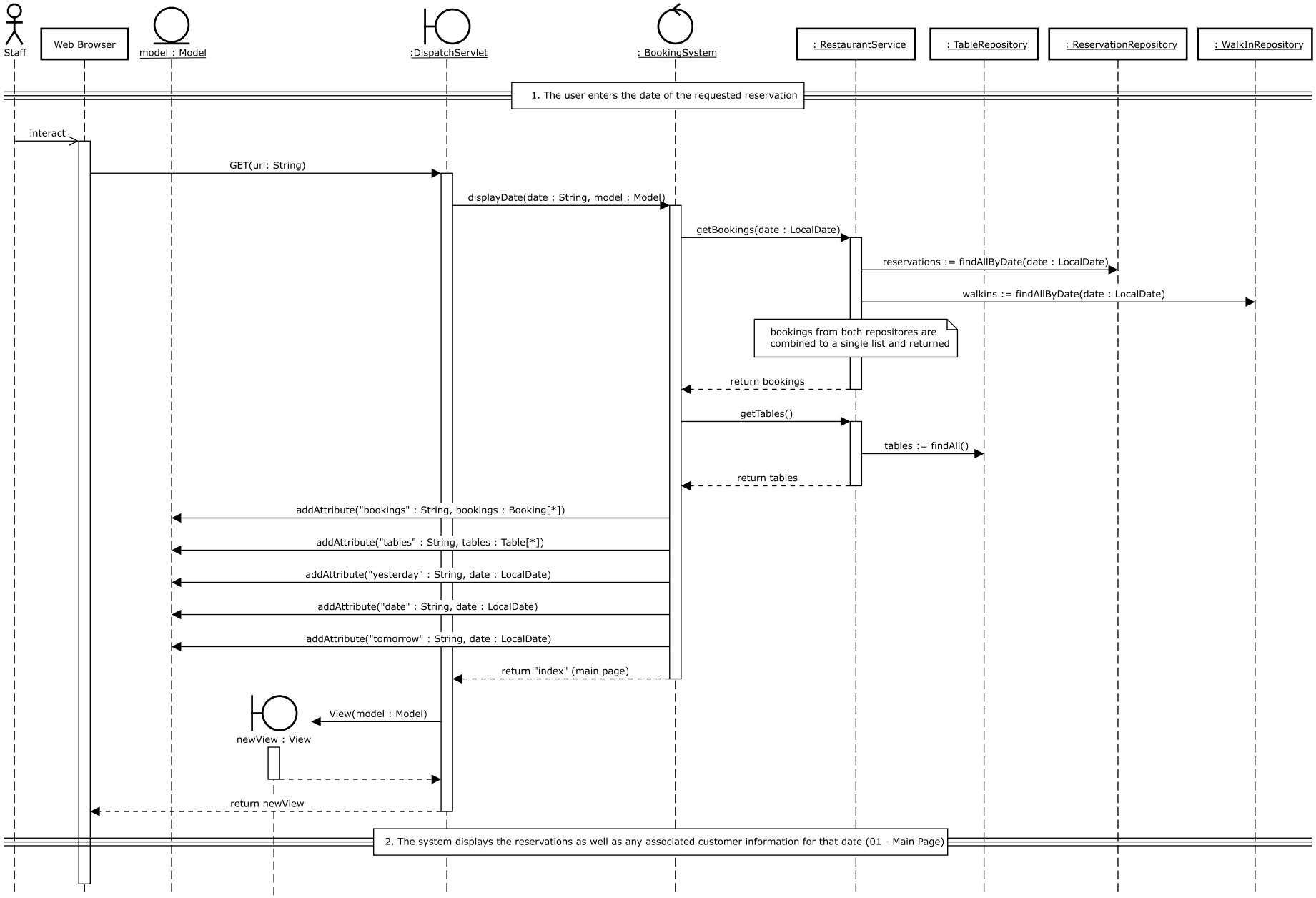
- Unique identifiers, we will need to add these to the classes
- Descriptions of the **relationships**
- WalkIn/Reservation and Table
  - A booking may be connected to only a single table
  - Each table may be connected to many bookings
- Reservation and Customer
  - A reservation may be connected to only a single customer
  - Each customer may be connected to many reservations

- For each persistent class, we will have a separate repository interface
- When we want to save a new object (or one we have updated), we should use the `save` method
- When we need to find an object, we should use one of the find methods

Section Contents

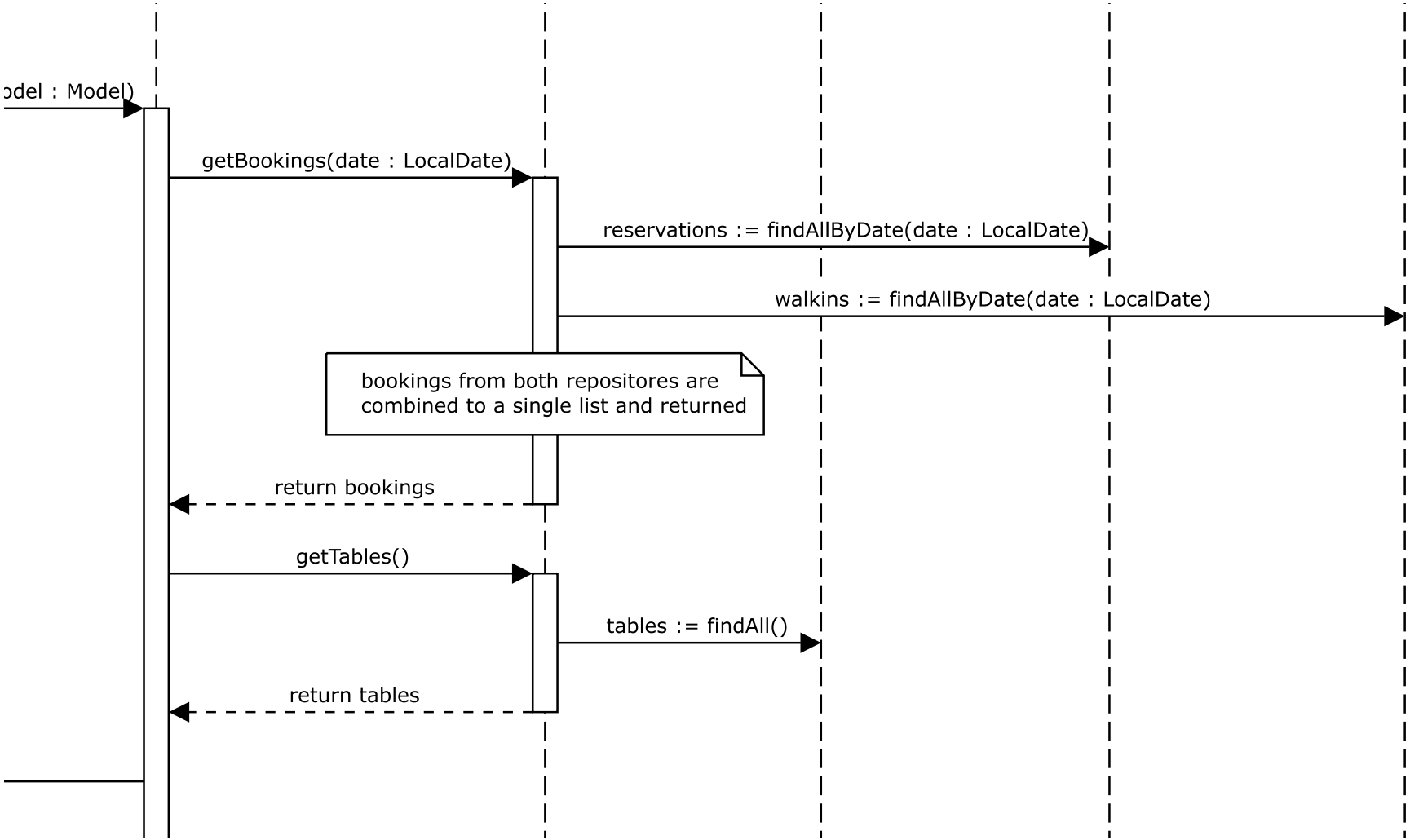
- Original sequence diagrams will have to be updated to include UI and database
  - Inclusion of DispatchServlet and Web Broswer
  - Inclusion of *Repository objects
  - Construction of Model for template

Display Date - Basic course of events

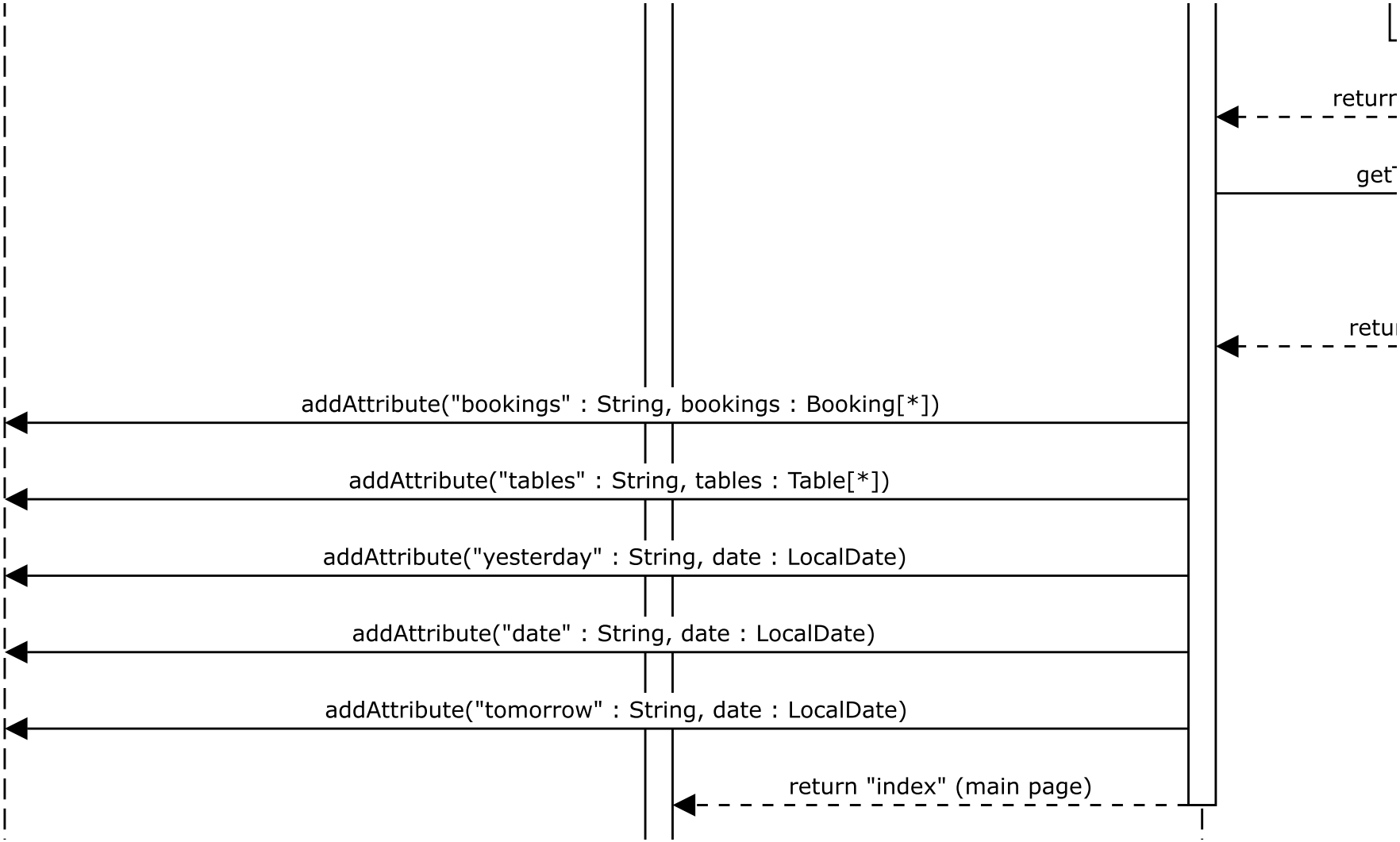


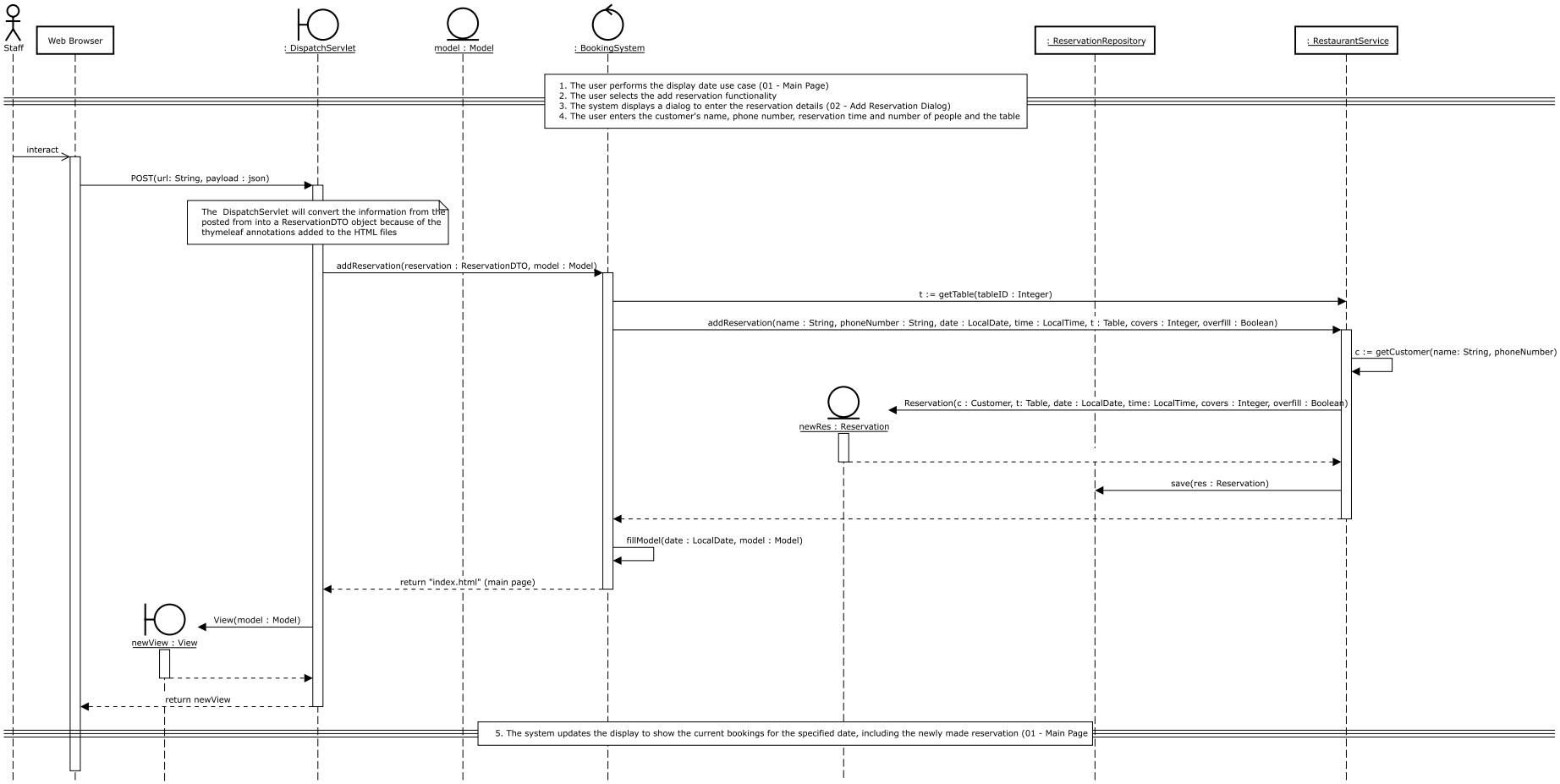
- For you assignment there are two important things that should be shown
  - How the necessary data is retrieved
  - That this data is added to the model
- In this example, the same data is required for every page
- As such it is done in a separate method for all of the remaining use cases

Getting the Data

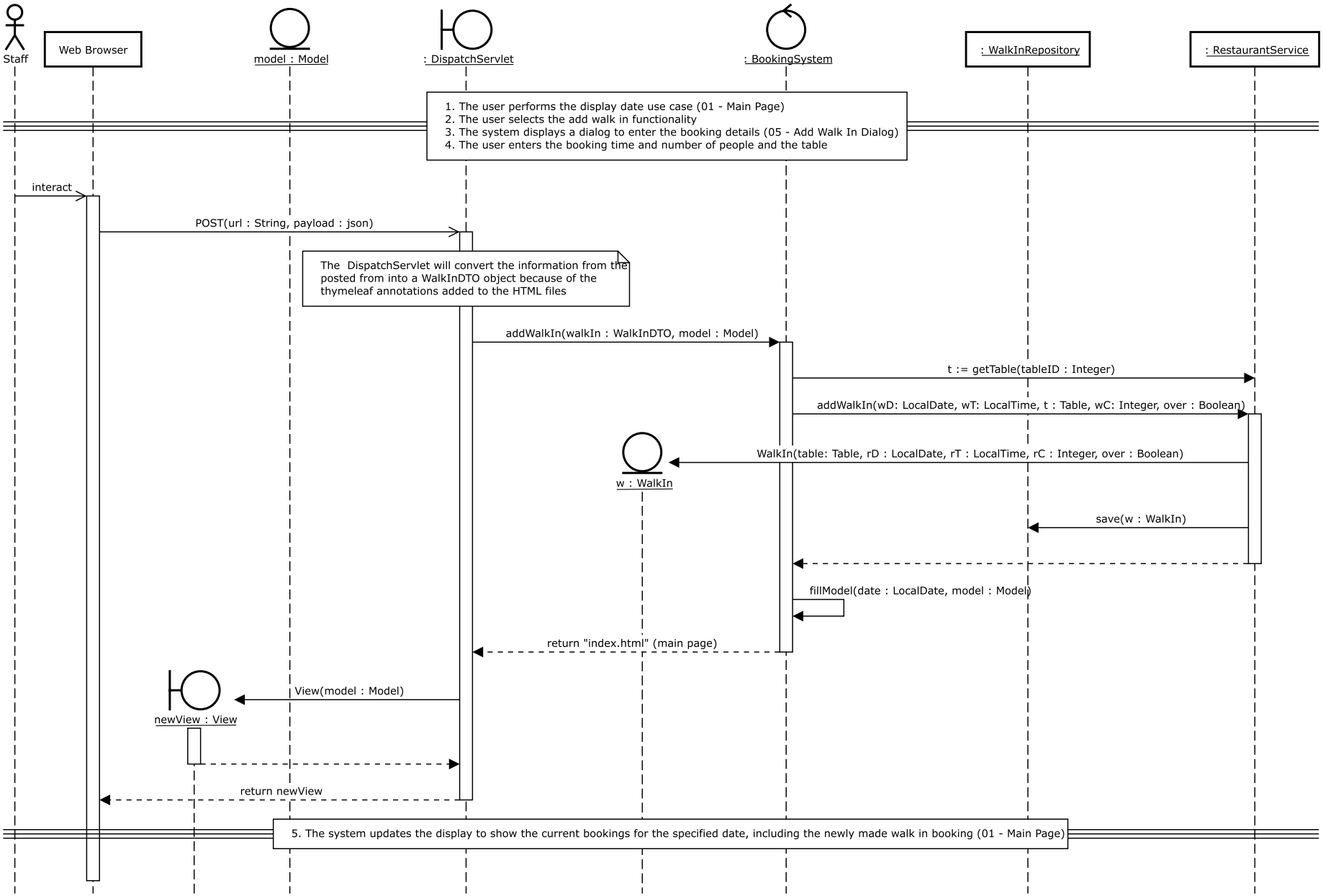


Data into Model

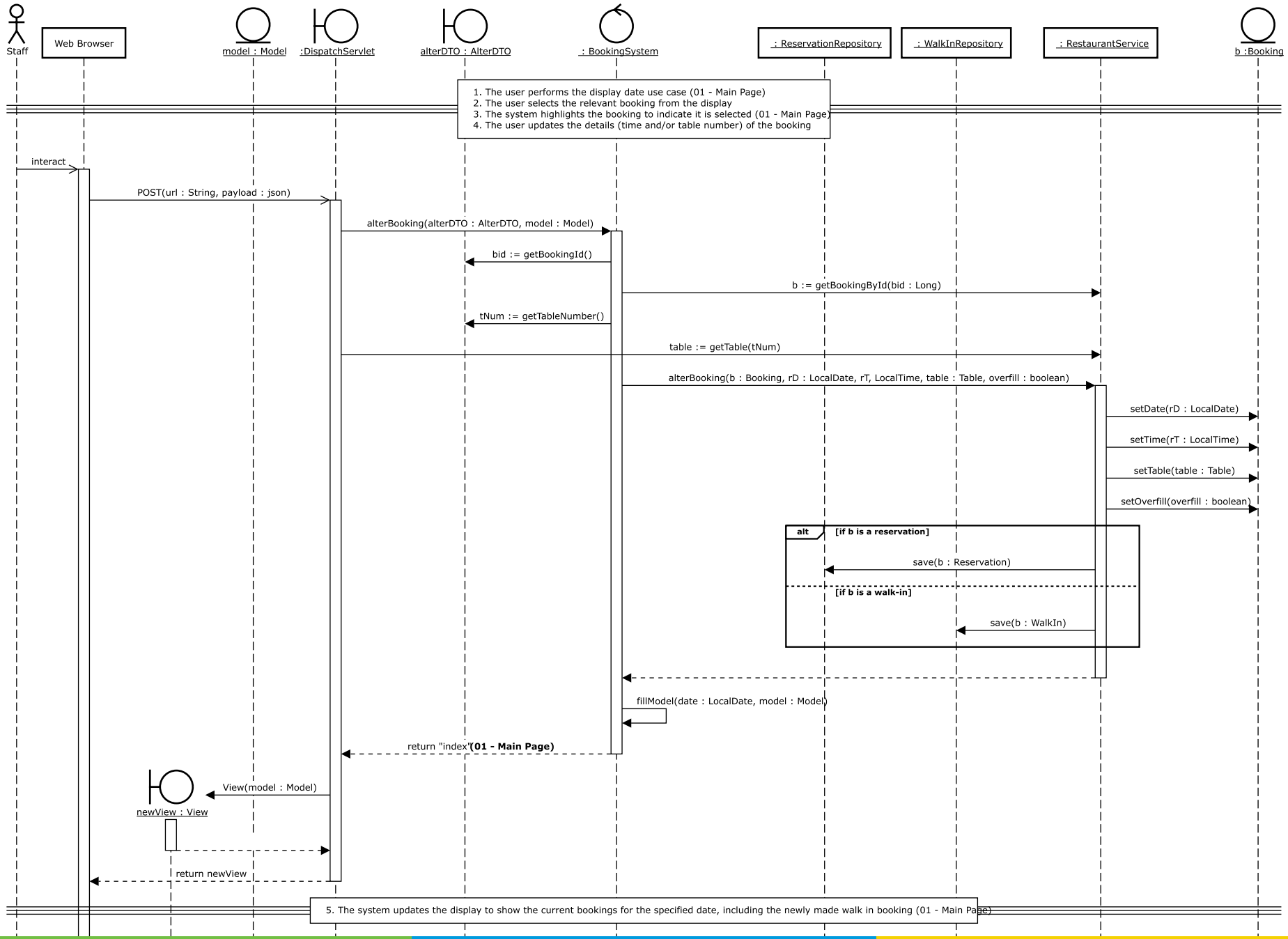



Staff
Web Browser
: DispatchServlet
model : Model
: BookingSystem
: ReservationRepository
: RestaurantService
1. The user performs the display date use case (01 - Main Page)
2. The user selects the add reservation functionality
3. The system displays a dialog to enter the reservation details (02 - Add Reservation Dialog)
4. The user enters the customer's name, phone number, reservation time and number of people and the table
interact
POST(url: String, payload : json)
The DispatchServlet will convert the information from the posted from into a ReservationDTO object because of the thymeleaf annotations added to the HTML files
addReservation(reservation : ReservationDTO, model : Model)
t := getTable(tableID : Integer)
addReservation(name : String, phoneNumber : String, date : LocalDate, time : LocalTime, t : Table, covers : Integer, overfill : Boolean)
c := getCustomer(name: String, phoneNumber)
Reservation(c : Customer, t: Table, date : LocalDate, time: LocalTime, covers : Integer, overfill : Boolean)
newRes : Reservation
save(res : Reservation)
fillModel(date : LocalDate, model : Model)
return "index.html" (main page)
View(model : Model)
newView : View
return newView
5. The system updates the display to show the current bookings for the specified date, including the newly made reservation (01 - Main Page

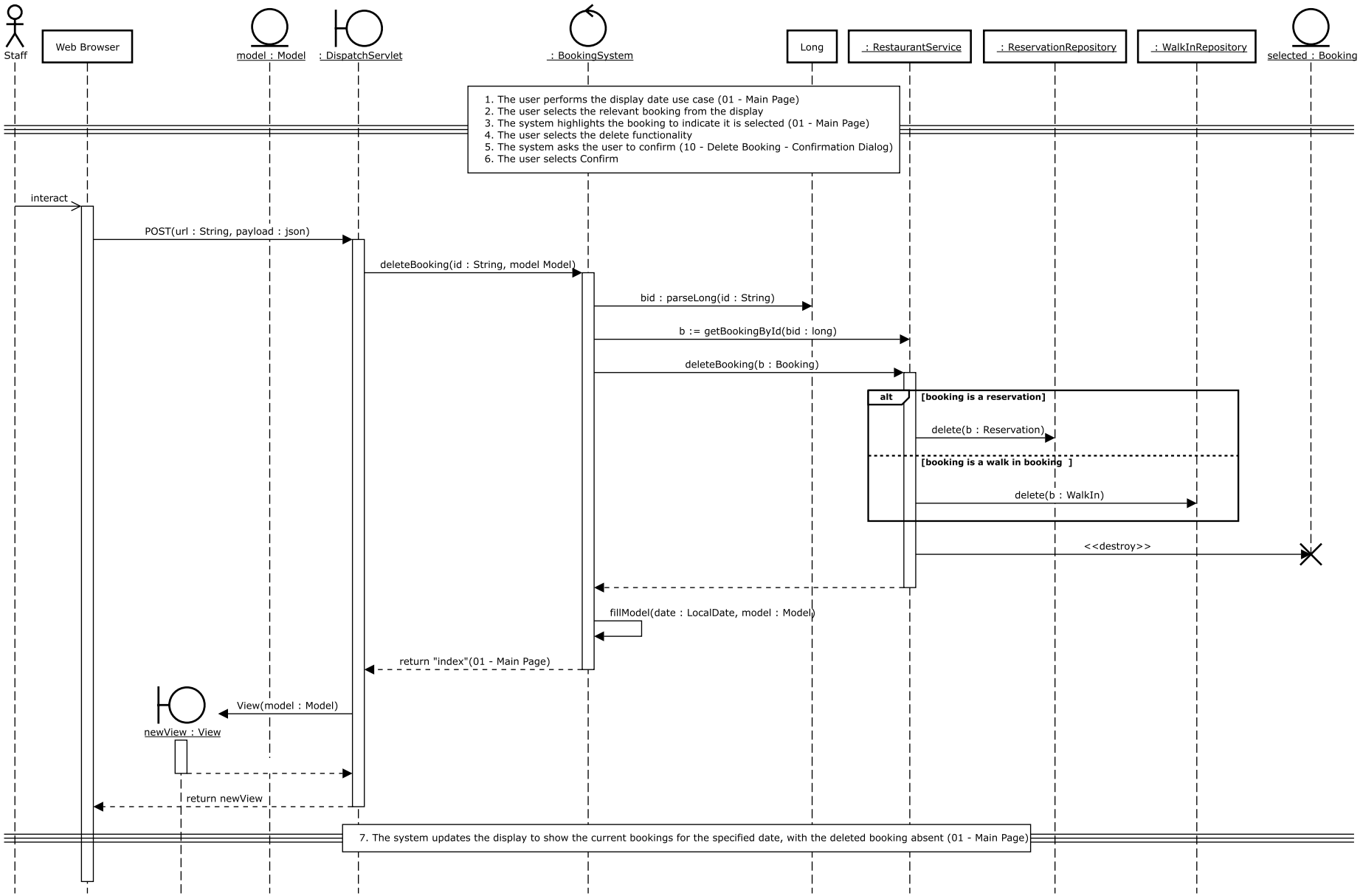
Add WalkIn - 01 Basic course of events



Alter Booking - 01 Basic Course of Events

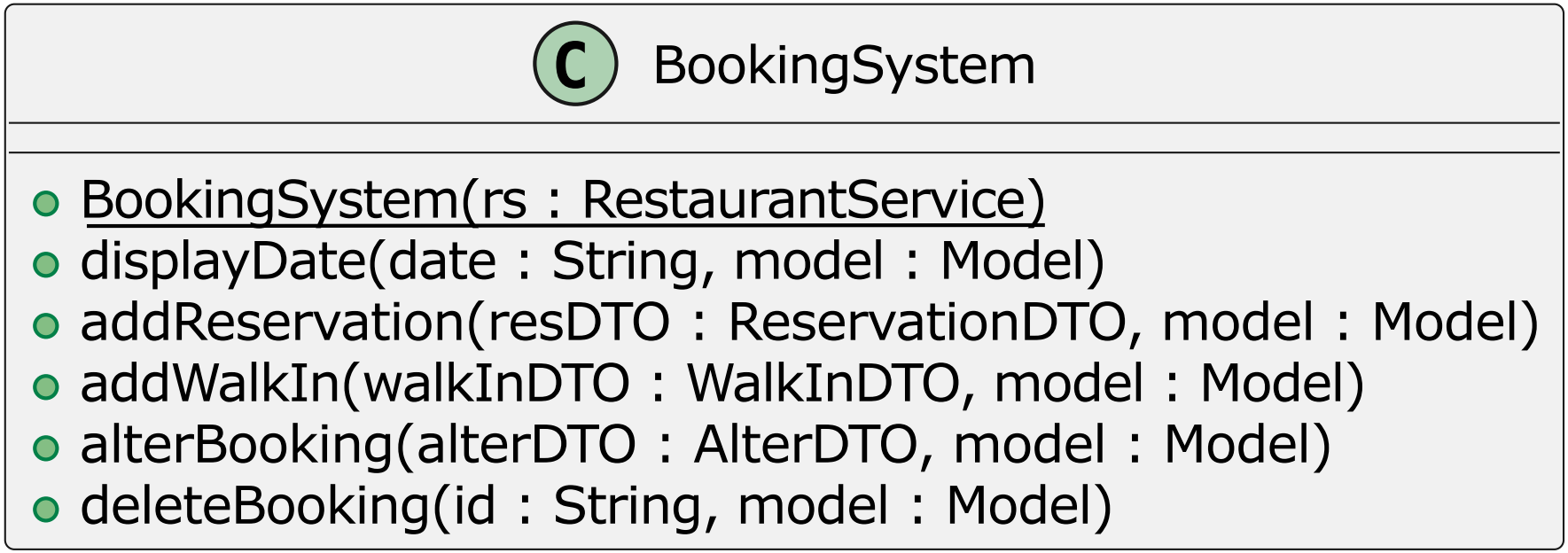


Delete Booking - 01 Basic Course of Events

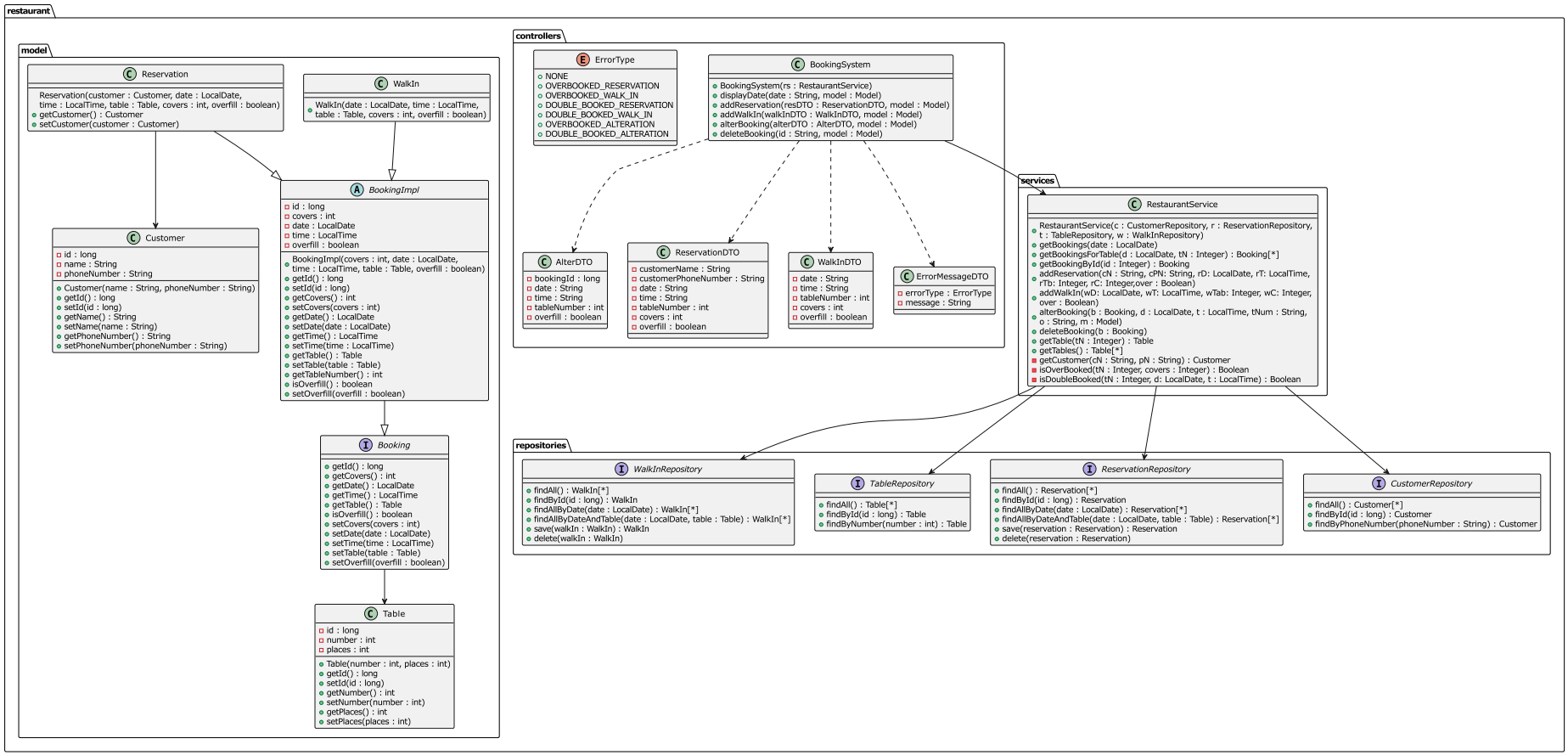


Section Contents

- As well as defining the overall structure of the system, a key design activity is to look in detail at the individual classes
- The sequence diagrams contain input to this activity, they define the messages that instances of classes must be able to respond to
- This therefore defined the operations that must be defined in each class
- This process proceeds by collecting all messages from the sequence diagrams, checking them for consistency and adding all information about parameters and return types as necessary


C
BookingSystem
BookingSystem(rs : RestaurantService)
displayDate(date : String, model : Model)
addReservation(resDTO : ReservationDTO, model : Model)
addWalkIn(walkInDTO : WalkInDTO, model : Model)
alterBooking(alterDTO : AlterDTO, model : Model)
deleteBooking(id : String, model : Model)


RestaurantService
RestaurantService(c : CustomerRepository, r : ReservationRepository, t : TableRepository, w : WalkInRepository)
getBookings(date : LocalDate)
getBookingsForTable(d : LocalDate, tN : Integer) : Booking[*]
getBookingById(id : Integer) : Booking
addReservation(cN : String, cPN: String, rD: LocalDate, rT: LocalTime, rTb: Integer, rC: Integer,over : Boolean)
addWalkIn(wD: LocalDate, wT: LocalTime, wTab: Integer, wC: Integer, over : Boolean)
alterBooking(b : Booking, d : LocalDate, t : LocalTime, tNum : String, o : String, m : Model)
deleteBooking(b : Booking)
getTable(tN : Integer) : Table
getTables() : Table[*]
getCustomer(cN : String, pN : String) : Customer
isOverBooked(tN : Integer, covers : Integer) : Boolean
isDoubleBooked(tN : Integer, d: LocalDate, t : LocalTime) : Boolean


restaurant
model
Reservation
Reservation(customer : Customer, date : LocalDate, time : LocalTime, table : Table, covers : int, overfill : boolean)
getCustomer() : Customer
setCustomer(customer : Customer)
WalkIn
WalkIn(date : LocalDate, time : LocalTime, table : Table, covers : int, overfill : boolean)
Customer
id : long
name : String
phoneNumber : String
Customer(name : String, phoneNumber : String)
getId() : long
setId(id : long)
getName() : String
setName(name : String)
getPhoneNumber() : String
setPhoneNumber(phoneNumber : String)
BookingImpl
id : long
covers : int
date : LocalDate
time : LocalTime
overfill : boolean
BookingImpl(covers : int, date : LocalDate, time : LocalTime, table : Table, overfill : boolean)
getId() : long
setId(id : long)
getCovers() : int
setCovers(covers : int)
getDate() : LocalDate
setDate(date : LocalDate)
getTime() : LocalTime
setTime(time : LocalTime)
getTable() : Table
setTable(table : Table)
getTableNumber() : int
isOverfill() : boolean
setOverfill(overfill : boolean)
Booking
getId() : long
getCovers() : int
getDate() : LocalDate
getTime() : LocalTime
getTable() : Table
isOverfill() : boolean
setCovers(covers : int)
setDate(date : LocalDate)
setTime(time : LocalTime)
setTable(table : Table)
setOverfill(overfill : boolean)
Table
id : long
number : int
places : int
Table(number : int, places : int)
getId() : long
setId(id : long)
getNumber() : int
setNumber(number : int)
getPlaces() : int
setPlaces(places : int)
controllers
ErrorType
NONE
OVERBOOKED_RESERVATION
OVERBOOKED_WALK_IN
DOUBLE_BOOKED_RESERVATION
DOUBLE_BOOKED_WALK_IN
OVERBOOKED_ALTERATION
DOUBLE_BOOKED_ALTERATION
BookingSystem
BookingSystem(rs : RestaurantService)
displayDate(date : String, model : Model)
addReservation(resDTO : ReservationDTO, model : Model)
addWalkIn(walkInDTO : WalkInDTO, model : Model)
alterBooking(alterDTO : AlterDTO, model : Model)
deleteBooking(id : String, model : Model)
AlterDTO
bookingId : long
date : String
time : String
tableNumber : int
overfill : boolean
ReservationDTO
customerName : String
customerPhoneNumber : String
date : String
time : String
tableNumber : int
covers : int
overfill : boolean
WalkInDTO
date : String
time : String
tableNumber : int
covers : int
overfill : boolean
ErrorMessageDTO
errorType : ErrorType
message : String
services
RestaurantService
RestaurantService(c : CustomerRepository, r : ReservationRepository, t : TableRepository, w : WalkInRepository)
getBookings(date : LocalDate)
getBookingsForTable(d : LocalDate, tN : Integer) : Booking[*]
getBookingById(id : Integer) : Booking
addReservation(cN : String, cPN: String, rD: LocalDate, rT: LocalTime, rTb: Integer, rC: Integer,over : Boolean)
addWalkIn(wD: LocalDate, wT: LocalTime, wTab: Integer, wC: Integer, over : Boolean)
alterBooking(b : Booking, d : LocalDate, t : LocalTime, tNum : String, o : String, m : Model)
deleteBooking(b : Booking)
getTable(tN : Integer) : Table
getTables() : Table[*]
getCustomer(cN : String, pN : String) : Customer
isOverBooked(tN : Integer, covers : Integer) : Boolean
isDoubleBooked(tN : Integer, d: LocalDate, t : LocalTime) : Boolean
repositories
WalkInRepository
findAll() : WalkIn[*]
findById(id : long) : WalkIn
findAllByDate(date : LocalDate) : WalkIn[*]
findAllByDateAndTable(date : LocalDate, table : Table) : WalkIn[*]
save(walkIn : WalkIn) : WalkIn
delete(walkIn : WalkIn)
TableRepository
findAll() : Table[*]
findById(id : long) : Table
findByNumber(number : int) : Table
ReservationRepository
findAll() : Reservation[*]
findById(id : long) : Reservation
findAllByDate(date : LocalDate) : Reservation[*]
findAllByDateAndTable(date : LocalDate, table : Table) : Reservation[*]
save(reservation : Reservation) : Reservation
delete(reservation : Reservation)
CustomerRepository
findAll() : Customer[*]
findById(id : long) : Customer
findByPhoneNumber(phoneNumber : String) : Customer