

## Memo

### Status

Not Complete. The reasons the project are not complete are listed below:

- The project meets all functional tests except for step 21, which states “Validate the seating in section sid1 for show wid1”. The project only updates the first row. If a patron orders a seat in another row, the seat availability will not change from available to sold.
- The makefile in the target folder can not find the TestRunner class, despite being in the same folder.
- There are some missing functionalities in the program. They include:
  - There is no occupancy or donated tickets report
  - One can only search for orders.
- There are some aspects of the program that are only designed to work with the functional testing script. They include:
  - The seat request algorithm that only works for rows that have 4 or less seats.
  - The ticketing system for the orders do not work past order 413.
  - The orders by date request only returns orders that were made in the same day.

### Code Repo

<https://github.com/jdicken2/Thalia>

### Lines of Code

Examples of a class from each package:

#### **DatabaseClass**

The database class was used as a mock database. It stored all data that was used in the project. The majority of the data structures were hashmaps.

```
public class DatabaseClass {  
    private static Map<Integer, Show> shows=new HashMap<>();  
    private static Map<Integer, Section> sections=new HashMap<>();  
    private static Map<Integer, InitialSeating> initialSeating=new HashMap<>();  
    private static Map<Integer, Order> orders=new HashMap<>();  
    //private static Map<Integer, Seats[]> seats=new HashMap<>();  
    private static Map<InitialSeating, ArrayList <Seating>> seating=new HashMap<>();  
    private static Map<Integer, FirstInitialSeating> firstInitialSeating=new HashMap<>();  
    private static Map<Integer, Donations> donations=new HashMap<>();  
    private static Map<Integer, Subscription> subscriptions=new HashMap<>();  
    private static Map<Integer, specificInitialSeating> specificInitialSeating=new HashMap<>();  
}
```

```
private static Map<Integer, specificOrder> specificOrders=new HashMap<>();
private static Ticket tickets=new Ticket();
private static Map<Integer, bodyPostOrder> bodyPostOrders=new HashMap<>();
private static ArrayList <specifiedReport> specifiedReports=new ArrayList <specifiedReport>();
private static Map<Integer, PostOrder> postOrders=new HashMap<>();

public static Map<Integer, FirstInitialSeating> getFirstInitialSeating()
{
    return firstInitialSeating;
}
public static Map<Integer, Show> getShow()
{
    return shows;
}
public static Map<Integer, Section> getSections()
{
    return sections;
}
public static Map<Integer, InitialSeating> getInitialSeating()
{
    return initialSeating;
}
public static Map<Integer, Order> getOrders()
{
    return orders;
}
public static Map<InitialSeating, ArrayList <Seating>> getSeating()
{
    return seating;
}

public static Map<Integer, Donations> getDonations()
{

```

```
        return donations;
    }

    public static Map<Integer, Subscription> getSubscriptions()
    {
        return subscriptions;
    }

    public static Ticket getTicket()
    {
        return tickets;
    }

    public static Map <Integer, specificInitialSeating> getSpecificInitialSeating()
    {
        return specificInitialSeating;
    }

    public static Map <Integer, specificOrder> getSpecificOrders()
    {
        return specificOrders;
    }

    public static Map <Integer, bodyPostOrder> getBodyPostOrders()
    {
        return bodyPostOrders;
    }

    public static ArrayList <specifiedReport> getSpecifiedReports()
    {
        return specifiedReports;
    }

    public static Map <Integer, PostOrder> getPostOrders()
    {
```

```
        return postOrders;
    }
}
```

### Resource Class Example

Several Resource Classes were used as my REST Controller. The Show Resource Class is an example of my general structure.

```
@Path("/shows")

public class ShowResource {

    private ShowBoundaryInterface sb=new ShowManager();

    public ShowResource()
    {
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAllShows()
    {
        Gson gson = new GsonBuilder().setPrettyPrinting().create();

        String s = gson.toJson(sb.getAllShows());

        return Response.status(Response.Status.OK).entity(s).build();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response postShow(String json)
    {
        Gson gson=new Gson();

        Show s=gson.fromJson(json, Show.class);

        gson.toJson(sb.addShow(s));

        Gson gson1=new GsonBuilder().setPrettyPrinting().create();
```

```
        String show=gson1.toJson(s);

        StringTokenizer str=new StringTokenizer(show, ",");

        String wid=str.nextToken();

        return Response.status(Response.Status.OK).entity(wid + '\n' + ").build();

    }

    @GET
    @Path("/{showID}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getShow(@PathParam("showID") int showID)
    {
        Show s=sb.getShow(showID);

        if(s.isNil())
        {
            return Response.status(Response.Status.NOT_FOUND).entity("Entity not found for ID: "
                + showID).build();
        }

        else {

            Gson gson = new GsonBuilder().setPrettyPrinting().create();

            String str=gson.toJson(s);

            return Response.ok(str).build();
        }
    }

    @PUT
    @Path("/{showID}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response putShow(@Context UriInfo uriInfo, @PathParam("showID") int showID, String json)
    {
        Gson gson=new Gson();

        Show s=gson.fromJson(json, Show.class);
```

```
        int id;

        s.setShowID(showID);

        id=s.getShowID();

        sb.updateShow(s);

        UriBuilder builder = uriInfo.getAbsolutePathBuilder();
        builder.path(Integer.toString(id));

String str=" ";
return Response.ok(str).build();
    }

    @GET
    @Path("/{showID}/sections")
    @Produces(MediaType.APPLICATION_JSON)
    public Response gotoSections()
    {
        SectionResource sr=new SectionResource();

        return sr.getSection();
    }

    @GET
    @Path("/{showID}/sections/{sectionID}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response gotoChairs(@PathParam("sectionID") int sectionID)
    {
        SectionResource sr=new SectionResource();

        return sr.getSeatingSection(sectionID);
    }

    @POST
```

```

@Path("/{showID}/donations")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response gotoDonations(String json)
{
    DonationResource dr=new DonationResource();
    return dr.postDonation(json);
}

@GET
@Path("/{showID}/donations/{donationID}")
@Produces(MediaType.APPLICATION_JSON)
public Response gotoSubscriptions(@PathParam("donationID") int donationID)
{
    DonationResource dr=new DonationResource();
    return dr.getSubscription(donationID);
}
}

```

### Boundary Interface Class

Boundary interfaces were used for managers to rely on interfaces and not actual classes. An example is shown below:

```

public interface ShowBoundaryInterface
{
    public int getWid();
    public ArrayList <someShowInfo> getAllShows();
    public Show getShow(int id);
    public Show addShow(Show sh);
    public Show updateShow(Show sh);
}

```

### Manager Class Example

Managers were used to interact with the database, such as returning one instance of a show, based on the show id.

```

public class ShowManager implements ShowBoundaryInterface

```

```
{

    public Map<Integer, Show> shows=DatabaseClass.getShow();
    private int wid=308;

    public ShowManager()
    {

    }

    @Override
    public int getWid()
    {
        return wid;
    }

    @Override
    public ArrayList <someShowInfo> getAllShows()
    {

        ArrayList <someShowInfo> si=new ArrayList <someShowInfo>();

        for(int i=0;i<shows.size();i++)
        {
            Show_Info sh=shows.get(wid).getShowInfo();
            someShowInfo shi=new someShowInfo(wid,sh);
            si.add(shi);
            wid++;
        }
        return si;
    }

    @Override
    public Show getShow(int id)
    {
        return shows.get(id);
    }

    @Override
    public Show addShow(Show sh)
    {
        sh.setShowID(wid+(shows.size()));
        shows.put(sh.getShowID(), sh);
        return sh;
    }

    @Override
    public Show updateShow(Show sh)
    {
        if(sh.getShowID()<=0)
        {
            return null;
        }
    }
}
```



```
        shows.put(sh.getShowID(), sh);
        return sh;
    }
}
```

## Service Class

Service classes were used to build the structure of several entities used throughout the project. The example shown is an example of a Show service class.

```
public class Show {

    private int wid;
    private Show_Info show_info;
    private List <Seating_Info> seating_info;
    public Show()
    {

    }

    public Show(int sID, Show_Info sh, List <Seating_Info> seat)
    {
        wid=sID;
        show_info=sh;
        seating_info=seat;
    }

    public int getShowID() {
        return wid;
    }

    public Show_Info getShowInfo() {
        return show_info;
    }

    public List <Seating_Info> getSeatingInfo()
    {
        return seating_info;
    }

    public void setShowID(int showID) {
        wid = showID;
    }

    public void setShowInfo(Show_Info showInfo) {
        show_info = showInfo;
    }

    public void setSeatingInfo(List <Seating_Info> seat)
    {
        seating_info=seat;
    }

    public boolean isNil() {
        return false;
    }
}
```

### Unit tests

Here are some examples of unit tests. They assert whether the expected or actual output are equivalent.

```
@Test
public void addShowTest()
{
    Show_Info si=new Show_Info("King Lear", "http://www.example.com/shows/king-
    lear", "2017-12-05", "13:00");
    ArrayList <Seating_Info> seatArray=new ArrayList <Seating_Info>();
    Seating_Info seat1=new Seating_Info(123, 60);
    Seating_Info seat2=new Seating_Info(124, 75);
    Seating_Info seat3=new Seating_Info(125, 60);
    seatArray.add(seat1);
    seatArray.add(seat2);
    seatArray.add(seat3);
    Show s=new Show(309,si,seatArray);
    Show sho=sb.addShow(s);
    assertEquals(s,sho);
}
```

```
@Test
public void updateShowTest()
{
    Show_Info si1=new Show_Info("King Lear", "http://www.example.com/shows/king-
    lear", "2017-12-05", "13:00");
    ArrayList <Seating_Info> seatArray1=new ArrayList <Seating_Info>();
    Seating_Info seat4=new Seating_Info(123, 60);
    Seating_Info seat5=new Seating_Info(124, 75);
    Seating_Info seat6=new Seating_Info(125, 60);
    seatArray1.add(seat4);
    seatArray1.add(seat5);
    seatArray1.add(seat6);
    Show s1=new Show(308, si1, seatArray1);
    Show returnShow=sb.updateShow(s1);
    assertEquals(s1,returnShow);
}
```



- Challenge: Starting the Project
  - Description: I had several issues understanding Apache Tomcat and RESTful Webservices. I did not know what to build my project on (Jersey/JAX RS).
  - Solution: I watched video tutorials on Jersey and JAX/RS
- Challenge: Testing the Project
  - Description: I wasn't sure how to test the project for core functionality. I am not familiar creating programs with no main method.
  - Solution: I used a REST Client for a browser, and eventually made the functional testing script work with my project.
- Challenge: Naming Conventions
  - Description: I did not know that the names of the actual classes did not matter. I assumed they did, and that's how JSON detected the names of the separate JSON sections.
  - Solution: I learned that the names of the classes do not matter, but before that I placed two of the classes in separate folders.