Software Engineering 2: myTaxiService

Integration Test Plan Document

Chitti Eleonora, De Nicolao Pietro, Delbono Alex
Politecnico di Milano

January 21, 2016

Contents

Chapter 1

Introduction

1.1 Revision History

Version of this document: 1.0
Last update: 21 January 2016

1.2 Purpose and Scope

1.2.1 Purpose

This document is the Integration Test Plan Document (ITPD) for the myTaxiService software. Its purpose is to determine how to accomplish the integration test of the software, which tools are to be used and which approach will be followed.

1.2.2 Scope

myTaxiService is a taxi reservation and dispatching system for large cities. Its goal is to simplify the access of passengers to the service and to guarantee a fair management of taxi queues.

1.3 List of Definitions and Abbreviations

RASD: Requirements Analysis and Specification Document.

DD: Design Document.

ITPD: Integration Test Plan Document (this document).

RDBMS: Relational Data Base Management System.

DB: the database layer, handled by a RDBMS.

Application server, business tier or back-end: the layer which provides the application logic and interacts with the DB and with the front-ends.

Front-end: the components which use the application server services, namely the web front-end and the mobile applications.

Web server: the component that implements the web-based front-end. It interacts with the application server and with the users' browsers.

JSF: JavaServer Faces.

1.4 List of Reference Documents

This document refers to the following documents:

- Project goals, schedules and rules of the Software Engineering 2 project [1]
- Assignment 4 - Test Document [2]
- Example Test Document [5]
- Requirement Analysis and Specification Document of the myTaxiService project [3]
- Design Document of the myTaxiService project [4]

Chapter 2

Integration Strategy

2.1 Entry Criteria

This section describes the prerequisites that need to be met *before* integration testing can be started.

All the classes and methods must pass thorough **unit tests** which should reasonably discover major issues in the structure of the classes or in the implementation of the algorithms. Unit tests should have a minimum coverage of 90% of the lines of code and should be run automatically at each build using JUnit. Unit testing is not in the scope of this document and will not be specified in further detail.

Moreover, **code inspection** has to be performed on all the code in order to ensure maintainability, respect of conventions and find possible issues which could increase the testers' effort in next testing phases. Code inspection must be performed using automated tools as much as possible: manual testing should be reserved for the most difficult features to test.

Finally, the **documentation** of all classes and functions, written using JavaDoc, has to be complete and up-to-date in order to be used as a reference for integration testing development. In particular, the public interfaces of each class and module should be well specified. Where necessary, a formal specification language can be used.

The following documents must be delivered before integration testing can begin:

- Requirement Analysis and Specification Document of myTaxiService
- Design Document of myTaxiService
- Integration Testing Plan Document (this document)

2.2 Elements to be Integrated

In the Design Document [4, p. 6] we outlined four major high-level components, corresponding to the tiers of the system, which – from now on – will be referred to as **subsystems**:

Database tier. This is the DBMS; it is not part of the software to be developed, but has to be integrated.

Business tier. This subsystem implements all the application logic and communicates with the front-ends.

Web tier. The web tier implements the web interface and communicates with the business tier and the client browsers.

Client tier. The client tier consists of desktop web browsers and of our mobile application.

The integration process of our software is performed on two levels.

1. integration of the different components (classes, Java Beans) inside the same subsystem;

2. integration of different subsystems.

The first step needs to be performed only for the component which contains the pieces of software that we are going to develop, namely the business tier, the mobile application in the client tier and part of the web tier (which, as stated in the DD, uses JSF to implement the web application).

2.3 Integration Testing Strategy

The integration strategy of choice is the **bottom-up approach**. This choice comes natural since we assume we already have the unit tests for the smallest components, so we can proceed from the bottom.

Moreover, the higher-level subsystems outlined in section 2.2 are well separated and loosely coupled since they correspond to different tiers; they also communicate through well-defined interfaces (REST API, HTTP), so they will not be hard to integrate at a later time. In this way it will be possible also to limit the stubs needed in order to accomplish the integration, because the specific components do not use the general ones, so they do not require stubs.

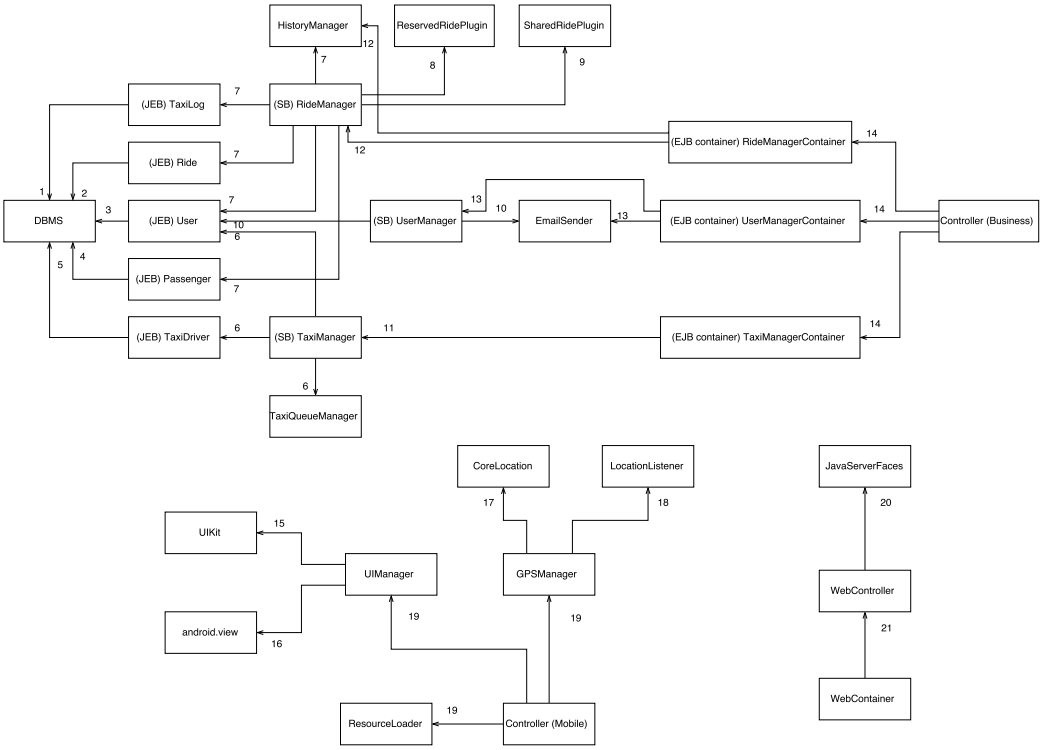

Figure 2.1: Diagram of the components integration.

2.4 Sequence of Component/Function Integration

2.4.1 Software Integration Sequence

The integration sequence of the components is described in Table 2.1 and in Figure 2.1.

The components are tested starting from the most independent to the less one. This gives the opportunity to avoid the implementation of useless stubs, because when less independent components are tested, the components which they rely on have already been integrated. The components are integrated within their classes in order to create an integrated subsystem which is ready for subsystem integration.

2.4.2 Subsystem Integration Sequence

The integration sequence of the subsystems is described in Table 2.2 and in Figure 2.2.

N.	Subsystem	Component	Integrates with
I1	Database, Business	(JEB) TaxiLog	DBMS
I2	Database, Business	(JEB) Ride	DBMS
I3	Database, Business	(JEB) User	DBMS
I4	Database, Business	(JEB) Passenger	DBMS
I5	Database, Business	(JEB) TaxiDriver	DBMS
I6	Business	(SB) TaxiManager	TaxiQueueManager TaxiDriver
I7	Business	(SB) RideManager	HistoryManager User Passenger TaxiLog Ride
I8	Business	(SB) RideManager	ReservedRidePlugin
I9	Business	(SB) RideManager	SharedRidePlugin
I10	Business	(SB) UserManager	EmailSender User
I11	Business	(EJB container) TaxiManagerContainer	TaxiManager
I12	Business	(EJB container) RideManagerContainer	RideManager HistoryManager
I13	Business	(EJB container) UserManagerContainer	UserManager EmailSender
I14	Business	Controller	TaxiManagerContainer RideManagerContainer UserManagerContainer ConfiguratorBean
I15	Mobile	UIManager	UIKit
I16	Mobile	UIManager	android.view
I17	Mobile	GPSManager	CoreLocation
I18	Mobile	GPSManager	LocationListener
I19	Mobile	Controller	UIManager GPSManager ResourceLoader
I20	Web	WebController	JavaServerFaces
I21	Web	WebContainer	WebController

Table 2.1: Integration of the system component.

N.	Subsystem	Integrates with
SI1	Business tier	Database tier
SI2	Mobile application	Business tier
SI3	Web tier	Business tier
SI4	Client browser	Web tier

Table 2.2: Integration order of the subsystems described in section 2.2.

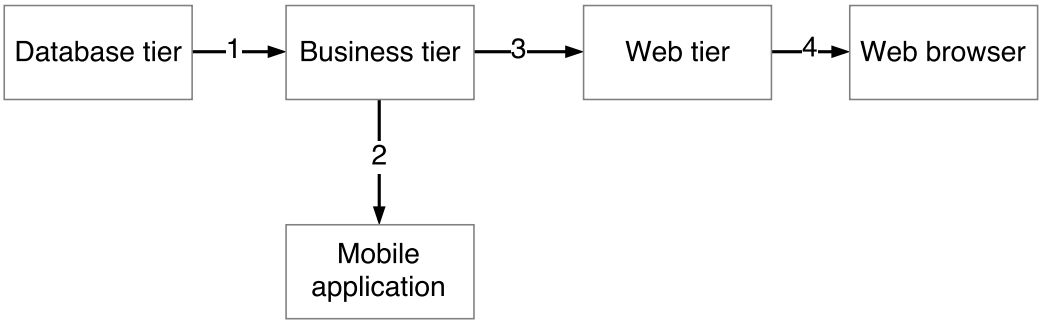

Figure 2.2: Directed Acyclic Graph representing the order of integration of the subsystems. See Table 2.2.

A choice was made to proceed with the integration process from the server side towards the client applications, integrating the mobile app before the web tier. The reason to do so is that in order to have a functioning client you need to have a working business tier. The business tier, instead, can be tested without any client, by making API calls also in an automated fashion.

By integrating the mobile application before the web tier, we aim to obtain a fully operational client-server system as soon as possible, since taxi drivers can not work without the mobile app. The web tier is less essential and can be integrated after the app.

Chapter 3

Individual Steps and Test Description

This chapter describes the individual test cases to be executed. Each test case is identified with a code and is directly mapped with Table 2.1 for the integration between components and with Table 2.2 for the integration between subsystems.

Test cases whose code starts with **SI** are integration tests between subsystems; test cases whose code starts with **I** are integration tests between components.

3.1 Integration test case SI1

Test Case Identifier	SI1T1
Test Item(s)	Business tier → Database Tier
Input Specification	Typical calls to the methods of the JPA Entities, mapped with tables in the Database tier.
Output Specification	The Database tier shall respond by doing the correct queries on the test database. It must also react in the right way both if the requests are made correctly and if they come from unauthorized sources that are trying to access the data.
Environmental Needs	Complete implementation of the Java Entity Beans, Java Persistence API, Test Database, driver that calls the Java Entity Beans.
Test Description	The response will be compared with the expected output of the queries.
Testing Method	Automated with JUnit.

3.2 Integration test case SI2

Test Case Identifier	SI2T1
Test Item(s)	Mobile application → Business Tier
Input Specification	Typical API calls (both correct and intentionally invalid ones) to the business tier (REST API).
Output Specification	The business tier shall respond accordingly to the API specification. Also, it must react correctly if the requests are malformed or maliciously crafted.
Environmental Needs	Complete implementation of the Business tier; REST API client (driver) that mocks the actual mobile client.
Test Description	The clients should make typical API calls to the business tier; the responses are then evaluated and checked against the expected output. The driver of this test is a standard REST API client that runs on Java.
Testing Method	Automated with JUnit.

Test Case Identifier	SI2T2
Test Item(s)	Mobile application → Business Tier
Input Specification	Multiple concurrent (typical, correct) requests to the REST API of the business tier.
Output Specification	The business tier must answer the requests in a reasonable time with the applied load.
Environmental Needs	GlassFish Server, fully developed business tier, Apache JMeter.
Test Description	This test case assesses whether the business tier fulfills the performance requirement stated in the RASD [3] (section 3.3, *Performance requirements*). In particular, the server has to support at least 1000 connected passengers at once, 95% of requests shall be processed in less than 5 s and 100% of requests shall be processed in less than 10 s.
Testing Method	Automated with Apache JMeter.

3.3 Integration test case SI3

Test Case Identifier	SI3T1
Test Item(s)	Web tier → Business tier
Input Specification	Requests for services offered by the business tier, also invalid ones.
Output Specification	The web tier must call the proper REST APIs or report an error.
Environmental Needs	GlassFish Server, Web tier.
Test Description	This test has to ensure the right translation from HTTPS requests into REST APIs calls, reporting errors when needed.
Testing Method	Automated with JUnit.

Test Case Identifier	SI3T2
Test Item(s)	Web tier → Business tier
Input Specification	Multiple concurrent API calls to the Business tier.
Output Specification	Web requests should be served without problems when a reasonable load is applied on the Business tier.
Environmental Needs	GlassFish Server, Web tier, Apache JMeter.
Test Description	This test case assesses whether the business tier fulfills the performance requirement stated in the RASD [3] (section 3.3, *Performance requirements*). In particular, the system has to support at least 1000 connected passengers at once, 95% of requests shall be processed in less than 5 s and 100% of requests shall be processed in less than 10 s.
Testing Method	Automated with Apache JMeter.

3.4 Integration test case SI4

Test Case Identifier	SI4T1
Test Item(s)	Client browser → Web tier
Input Specification	Typical and well-formed HTTPS requests from client browser; incomplete, malformed and maliciously crafted requests.
Output Specification	The web tier shall display the requested pages if the requests are valid; if the requests are invalid it shall display a generic error message.
Environmental Needs	GlassFish Server, fully developed web tier, HTTP client (driver).
Test Description	This test should emulate HTTP requests from typical users of the service and also incorrect requests.
Testing Method	Automated with JUnit.

Test Case Identifier	SI4T1
Test Item(s)	Client browser → Web tier
Input Specification	Multiple concurrent requests to the web server.
Output Specification	Web pages should be served without problems when a reasonable load is applied on the web server.
Environmental Needs	GlassFish Server, fully developed web tier, Apache JMeter.
Test Description	This test case assesses whether the web tier fulfills the performance requirement stated in the RASD [3] (section 3.3, *Performance requirements*). In particular, the web tier has to support at least 1000 connected passengers at once, 95% of requests shall be processed in less than 5 s and 100% of requests shall be processed in less than 10 s.
Testing Method	Automated with Apache JMeter.

3.5 Integration test cases I01, I02, I03, I04, I05: components that integrate with the DBMS

The following test cases refer to the integration between the Java Entity Beans and the underlying Database tier. Since the test cases are very similar, they are grouped together.

Test Case Identifier	I01T1
Test Item(s)	TaxiLog → DBMS
Input Specification	Typical queries on table TaxiLog.
Test Case Identifier	I02T1
Test Item(s)	Ride → DBMS
Input Specification	Typical queries on table Ride.
Test Case Identifier	I03T1
Test Item(s)	User → DBMS
Input Specification	Typical queries on table User.
Test Case Identifier	I04T1
Test Item(s)	Passenger → DBMS
Input Specification	Typical queries on table Passenger.
Test Case Identifier	I05T1
Test Item(s)	TaxiDriver → DBMS
Input Specification	Typical queries on table TaxiDriver.
Output Specification	The queries return the correct results.
Environmental Needs	GlassFish server, Test Database, driver for the Java Entity Beans.
Test Description	The purpose of these tests is to check that the correct methods of the Entity Beans are called, and that they execute the correct queries to the DBMS.
Testing Method	Automated with JUnit.

3.6 Integration test case I6

Test Case Identifier	I6T1
Test Item(s)	TaxiManager → TaxiQueueManager, TaxiDriver
Input Specification	Methods call from TaxiManager to TaxiQueueManager, to update driver's status and position and to find an available taxi in a specified TaxiZone.
Output Specification	The driver's position must be correctly updated without duplicating elements and the correct first available taxi must be returned and removed from the queue. The management of the driver's status must be properly handled.
Environmental Needs	GlassFish Server.
Test Description	The test aims to verify that the TaxiManager requests are correctly satisfied by TaxiQueueManager.
Testing Method	Automated with JUnit.

3.7 Integration test case I7

Test Case Identifier	I7T1
Test Item(s)	RideManager → HistoryManager, User, Passenger, TaxiLog, Ride
Input Specification	Methods call from RideManager to HistoryManager, to manage and update the information of the rides.
Output Specification	The rides information must be correct and up-to-date.
Environmental Needs	GlassFish Server.
Test Description	Verify that the information is correctly updated and that it refers to the correct ride. Control that the rides' information is persistently updated.
Testing Method	Automated with JUnit.

3.8 Integration test case I8

Test Case Identifier	I8T1
Test Item(s)	RideManager → ReservedRidePlugin
Input Specification	Calls to plugin methods to assure correct integration of the plugin.
Output Specification	The new functionalities of the plugin must be properly offered.
Environmental Needs	GlassFish Server.
Test Description	Assure that a ride can be reserved for a future time.
Testing Method	Automated with JUnit.

3.9 Integration test case I9

Test Case Identifier	I9T1
Test Item(s)	RideManager → SharedRidePlugin
Input Specification	Calls to plugin methods to assure correct integration of the plugin.
Output Specification	The new functionalities of the plugin must be properly offered.
Environmental Needs	GlassFish Server.
Test Description	Assure that a ride can be shared between multiple users and that a split fee is correctly computed.
Testing Method	Automated with JUnit.

3.10 Integration test case I10

Test Case Identifier	I10T1
Test Item(s)	UserManager → EmailSender, User
Input Specification	Methods call from UserManager to the EmailSender in order to guarantee a right email authentication process.
Output Specification	The email authentication process must be correctly handled.
Environmental Needs	GlassFish Server, mocked e-mail sender and receiver.
Test Description	Assure that a user can properly verify his/her email address in order to start using the system functionalities. In order to do that, a mock email address manager which simulates the user behaviour is needed.
Testing Method	Automated with JUnit and Mockito.

3.11 Integration test case I11

Test Case Identifier	I11T1
Test Item(s)	TaxiManagerContainer → TaxiManager
Input Specification	Requests for the TaxiManager SessionBeans.
Output Specification	The SessionBeans must be correctly assigned and the concurrency between the request must be properly managed.
Environmental Needs	GlassFish Server.
Test Description	Multiple requests for the TaxiManager SessionBeans have to be simultaneously carried out, in order to ensure that the users have no concurrency trouble.
Testing Method	Automated with JUnit and Arquillian.

3.12 Integration test case I12

Test Case Identifier	I12T1
Test Item(s)	RideManagerContainer → RideManager
Input Specification	Requests for the RideManager SessionBeans.
Output Specification	The SessionBeans must be correctly assigned and the concurrency between the request must be properly managed.
Environmental Needs	GlassFish Server.
Test Description	Multiple requests for the RideManager SessionBeans have to be simultaneously carried out, in order to ensure that the users have no concurrency trouble.
Testing Method	Automated with JUnit and Arquillian.

3.13 Integration test case I13

Test Case Identifier	I13T1
Test Item(s)	UserManagerContainer → UserManager
Input Specification	Requests for the UserManager SessionBeans.
Output Specification	The SessionBeans must be correctly assigned and the concurrency between the request must be properly managed.
Environmental Needs	GlassFish Server.
Test Description	Multiple requests for the UserManager SessionBeans have to be simultaneously carried out, in order to ensure that the users have no concurrency trouble.
Testing Method	Automated with JUnit and Arquillian.

3.14 Integration test case I14

Test Case Identifier	I14T1
Test Item(s)	Controller → TaxiManagerContainer, RideManagerContainer, UserManagerContainer
Input Specification	Requests from Controller to the containers for the functionalities offered by SessionBeans within containers.
Output Specification	The controller has to be able to provide the right functionality carrying out the proper request to the containers.
Environmental Needs	GlassFish Server.
Test Description	Ensure that the controller is able to provide the functionalities of the system offered by the containers.
Testing Method	Automated with JUnit and Arquillian.

3.15 Integration test case I15

Test Case Identifier	I15T1
Test Item(s)	UIManager → UIKit
Input Specification	Methods call from UIManager to the UI elements, to display output data and change their status.
Output Specification	The view shall change accordingly and display the output data.
Environmental Needs	Xcode, iOS Simulator.
Test Description	Verify that the bindings of the view items are correctly set in the controller and that the view actually changes and responds to method calls. Check that the output is displayed correctly.
Testing Method	Automated (iOS testing suite), manual testing on physical devices.

Test Case Identifier	I15T2
Test Item(s)	UIManager → UIKit
Input Specification	Perform (or simulate) gestures on the UI elements.
Output Specification	The controller shall receive the actions and log them.
Environmental Needs	Xcode, iOS Simulator.
Test Description	Check that the gestures perform the correct actions on the controller.
Testing Method	Automated (iOS testing suite), manual testing on physical devices.

3.16 Integration test case I16

Test Case Identifier	I16T1
Test Item(s)	UIManager → android.view
Input Specification	Methods call from UIManager to the UI elements, to display output data and change their status.
Output Specification	The view shall change accordingly and display the output data.
Environmental Needs	Android Emulator.
Test Description	Verify that the bindings of the view items are correctly set in the controller and that the view actually changes and responds to method calls. Check that the output is displayed correctly.
Testing Method	Automated (Android testing suite), manual testing on physical devices.

Test Case Identifier	I16T2
Test Item(s)	UIManager → android.view
Input Specification	Perform (or simulate) gestures on the UI elements.
Output Specification	The controller shall receive the actions and log them.
Environmental Needs	Android Emulator.
Test Description	Check that the gestures perform the correct actions on the controller.
Testing Method	Automated (Android testing suite), manual testing on physical devices.

3.17 Integration test case I17

Test Case Identifier	I17T1
Test Item(s)	GPSManager → CoreLocation
Input Specification	Calls to the CoreLocation framework methods to get location data of the user.
Output Specification	User location data or a meaningful error status shall be returned.
Environmental Needs	Xcode, iOS Simulator.
Test Description	The purpose of the test is to check that our controller (GPSManager) can correctly get the position from the corresponding iOS API. Error statuses shall also be checked.
Testing Method	Automated (iOS testing suite).

3.18 Integration test case I18

Test Case Identifier	I18T1
Test Item(s)	GPSManager → LocationListener
Input Specification	Calls to the Android Location framework methods to get location data of the user.
Output Specification	User location data shall be returned, or a meaningful error status.
Environmental Needs	Android Emulator.
Test Description	The purpose of the test is to check that our controller (GPSManager) can correctly get the position from the corresponding Android API. Error statuses shall also be checked.
Testing Method	Automated (Android testing suite).

3.19 Integration test case I19

Test Case Identifier	I19T1
Test Item(s)	UIManager → GPSManager
Input Specification	Calls to GPSManager methods to get the user's location.
Output Specification	The location data shall be returned from GPSManager in a suitable format, or an exception shall be raised if the location data is not available.
Environmental Needs	Xcode, iOS Simulator, Android Emulator.
Test Description	GPSManager should be able to return the correct GPS data in a universal and consistent format independently from the architecture (iOS or Android).
Testing Method	Automated (Android and iOS testing suites).

Test Case Identifier	I19T2
Test Item(s)	UIManager → ResourceLoader
Input Specification	Load application resources (images, sounds, data) from ResourceManager.
Output Specification	ResourceManager should provide the required resources without errors.
Environmental Needs	Xcode, iOS Simulator, Android Emulator.
Test Description	ResourceLoader is responsible for the retrieval of the resources stored into the application bundle. This test aims to assessing that all the resources can be accessed without errors by the mobile application.
Testing Method	Automated (Android and iOS testing suites).

3.20 Integration test case I20

Test Case Identifier	I20T1
Test Item(s)	WebController → JavaServerFaces
Input Specification	WebController is given the typical output to be displayed on the web page.
Output Specification	JavaServerFaces shall display the required output in a correct way.
Environmental Needs	GlassFish Server, Stub of the Business Tier to provide the output data.
Test Description	The purpose of this test case is to check if JSF can communicate correctly with the WebController bean.
Testing Method	Automated with JUnit.

3.21 Integration test case I21

Test Case Identifier	I21T1
Test Item(s)	WebContainer → WebController
Input Specification	Run the web application.
Output Specification	WebContainer injects the WebController bean, using JSF.
Environmental Needs	GlassFish Server.
Test Description	This test verifies if the correct component is injected into JSF.
Testing Method	Automated with JUnit.

Chapter 4

Tools and Test Equipment Required

The software tools used to automate the integration testing are the following:

Apache JMeter JMeter[1] is a powerful tool which may be used to test the performance of subsystems:

Web tier: simulate a heavy load on the web tier in order to check if the requirements on the maximum number of simultaneously connected users and on the response times stated in the RASD [3, p. 57] are respected. Performance testing on the web tier is described in section 3.3.

Business tier: simulate a heavy load on the REST API. Please note that a stress test on the web tier as described before can also overload the business tier; tests on both sides are useful to identify the bottlenecks. Performance testing on the business tier is described in section 3.2.

JUnit JUnit[2] is the most used framework for unit testing in Java. We plan to use it for unit tests of the single components (not covered by this document), but it is also used to do integration testing together with Mockito and Arquillian.

Arquillian Arquillian[3] is a test framework which can also manage the test of the containers and their integration with JavaBeans (dependency injection). We mainly use it for that purpose.

[1] `http://jmeter.apache.org/`
[2] `http://junit.org/`
[3] `http://arquillian.org/`

Mockito Mockito[4] is an open-source test framework useful to generate mock objects, stubs and drivers. We use it in several test cases to mock stubs and drivers for the components to test.

[4] `https://en.wikipedia.org/wiki/Mockito`

Chapter 5

Program Stubs and Test Data Required

In order to perform integration testing without having developed the entire system first ("big bang" approach), we need to use stubs and drivers to take the part of the software components that still don't exist and test the others.

Test database: the testing environment must include a DBMS configured in the same way of the production. The test data contained in this database includes a reduced set of instances of all the entities described in the Entity-Relation diagram of the Design Document [4, p. 10], which is reported in Figure 5.1.

Lightweight API client: in order to test the REST API of the business tier without the actual client application, a simple API client which interacts with the business tier by simple HTTP requests is needed. This driver needs to be scriptable in order for the tests to be automated.

Drivers for the Java Entity Beans: they are used to test the Java Entity Beans when the Business Tier is not fully developed. They call the relevant methods of the EJBs to test the correctness of the queries.

Stub of the Business Tier: used to provide a minimum set of data to test the web tier when the business tier is not fully developed.

Mock e-mail sender and receiver: to automate the testing of the e-mail confirmation process.

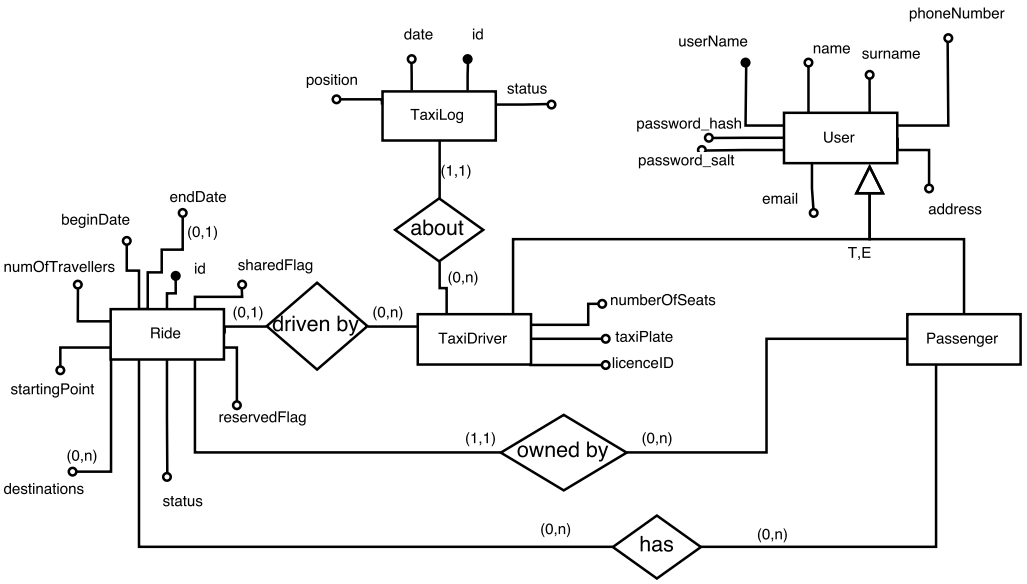

Figure 5.1: ER diagram of the database schema as specified in the Design Document [4].

Appendix A

Appendix

A.1 Software and tools used

- LaTeX for typesetting this document.
- GitHub[1] for version control and distributed work.

A.2 Hours of work

The statistics about commits and code contribution are available on GitHub [2]. Please keep in mind that many commits are actually group work (when this is the case, it is stated in the commit message).

- Eleonora Chitti: 5 hours
- Alex Delbono: 6 hours
- Pietro De Nicolao: 5 hours

[1] `https://github.com`
[2] `https://github.com/pietrodn/se2-mytaxiservice`

Bibliography

[1] Software Engineering 2 Project, AA 2015/2016, *Project goal, schedule and rules*

[2] Software Engineering 2 Project, AA 2015/2016, *Assignment 4: Test plan*

[3] Chitti Eleonora, De Nicolao Pietro, Delbono Alex, *Software Engineering 2: myTaxiService – Requirements Analysis and Specification Document*

[4] Chitti Eleonora, De Nicolao Pietro, Delbono Alex, *Software Engineering 2: myTaxiService - Design Document*

[5] SpinGrid Project - *Integration Test Plan*