

Interplanetary Space Transport System Design Document

Date: 12/18/2019

Author: Gerald Arocena

Reviewer(s): Krithika Sundararajan

Table of Contents

Interplanetary Space Transport System (ISTS) Architectural Overview.....	4
Introduction.....	4
Overview.....	4
Component Diagram.....	5
Requirements.....	6
Use Cases.....	6
Activity Diagram.....	6
Implementation Details.....	7
ISTS Resource Management Service Design Specification.....	8
Introduction.....	8
Overview.....	8
Requirements.....	8
Use Cases.....	9
Use Case Diagram.....	9
Implementation.....	11
Class Diagram.....	11
Class Dictionary.....	12
Implementation Details.....	25
Exception Handling.....	25
Testing.....	25
Risks.....	26
Abstractness and Instability Metrics.....	26
ISTS Customer Service Design Specification.....	27
Introduction.....	27
Overview.....	27
Requirements.....	28
Use Cases.....	28
Use Case Diagram.....	29
Implementation.....	31
Class Diagram.....	32
Class Dictionary.....	33
Implementation Details.....	49
Sequence Diagram (Book Flight).....	51
Exception Handling.....	52
Testing.....	52
Risks.....	52
Abstractness and Instability Metrics.....	53
ISTS Flight Management Service Design Specification.....	54
Introduction.....	54

Overview.....	54
Requirements.....	54
Use Cases.....	55
Use Case Diagram.....	55
Implementation.....	57
Class Diagram.....	57
Class Dictionary.....	58
Implementation Details.....	69
Sequence Diagram (Provision Flight).....	70
Sequence Diagram (Reached Destination event).....	71
Exception Handling.....	71
Testing.....	72
Risks.....	72
Abstractness and Instability Metrics.....	72
GUI.....	73
Customer Service GUI.....	73
Resource Management Service GUI.....	81
Flight Management Service GUI.....	84

Interplanetary Space Transport System (ISTS) Architectural Overview

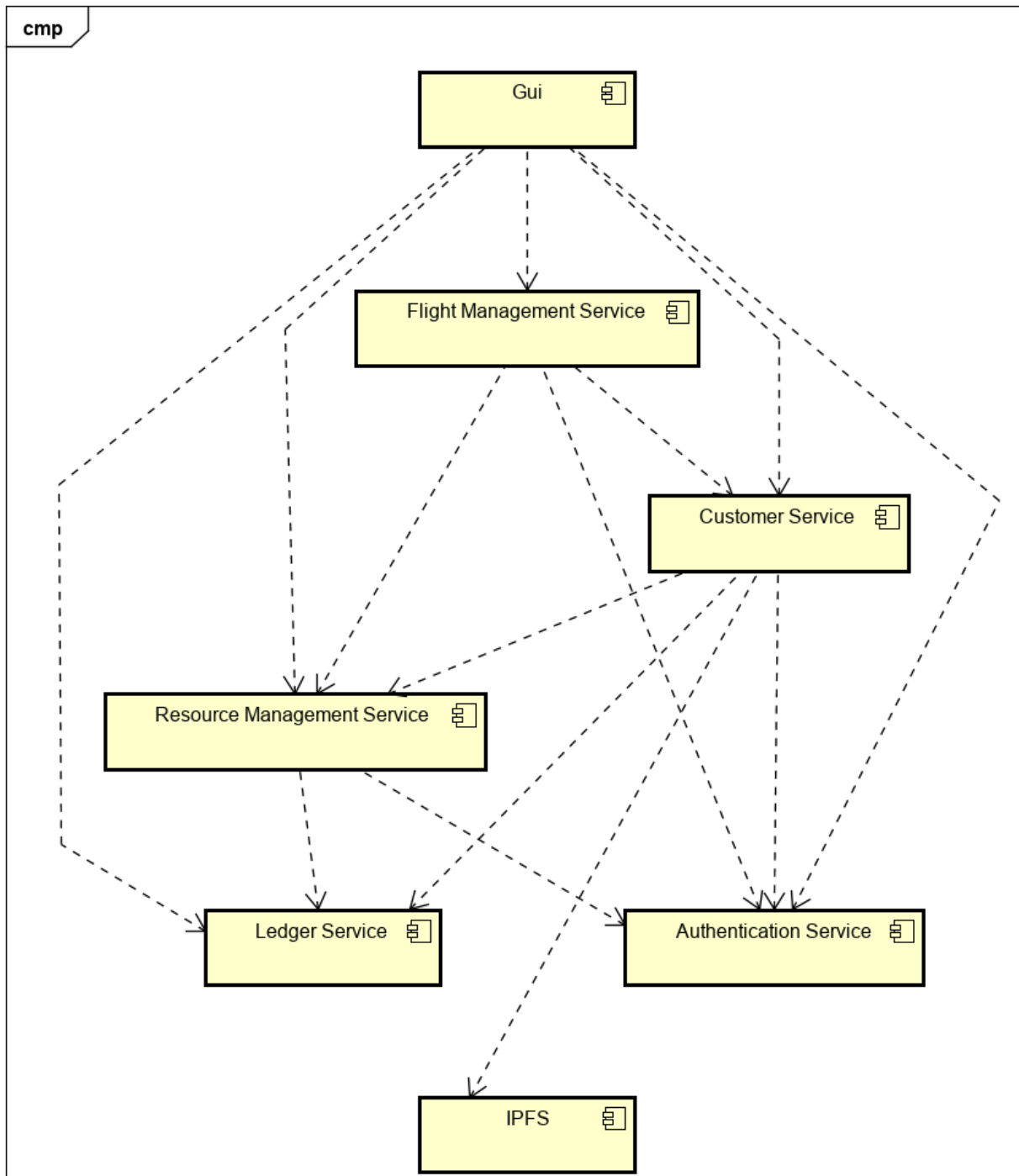
Introduction

This system design document explains how to implement the Interplanetary Space Travel System; a program put forth by the International Space Agency. The program's objective includes expanding knowledge of the solar system and the universe. The software system that this design document outlines includes the use of spaceships, communication systems, and distributed file systems.

Overview

Due to advances in space travel technology, humans now have the capability to travel through the universe and research it which has many benefits. For instance, space exploration can potentially save Earth from being victim to outerspace threats like asteroid impacts. In addition, exploring space can be seen as a leisure activity.

A high-level component diagram is shown on the next page that illustrates the relationship between the components that make up the Interplanetary Space Transport (IST) System.



Caption: The three main modules – Resource Management service, Customer service, and Flight Management service – are shown above as well as the other system components of Ledger service, Authentication service, IPFS (InterPlanetary File System), and GUI (graphical user interface). The arrows point from each component to their dependencies.

Requirements

The IST system must comprise three main modules: Resource Management, Customer service, and Flight Management. It must also provide a graphical user interface for each. The GUI should allow administrators and passengers to login to access the IST system and should be implemented using the underlying services. They must support the use cases (discussed below) in this document. In addition, the IST system must use the Ledger service blockchain system implemented in Assignment 1 to process any payment transactions including ticket purchases and ISTS account management. The Authentication service implemented in Assignment 4 must also be used to check access permissions for the GUI, service APIs, access to spaceships and when boarding passengers. Furthermore, the InterPlanetary File System (IPFS) must be used for storage for flight-related activities including for travel documents, in-flight entertainment, and documentation of flight experiences.

The modules will also need to be implemented independently of one another such that they exhibit level 5 of the Modularity Maturity Model: Service Oriented Architecture. Each module must define a service interface that the other modules can access and the GUI should use the service interfaces for access by users. Additionally, design patterns should be considered for use and highlighted where appropriate.

Persistence

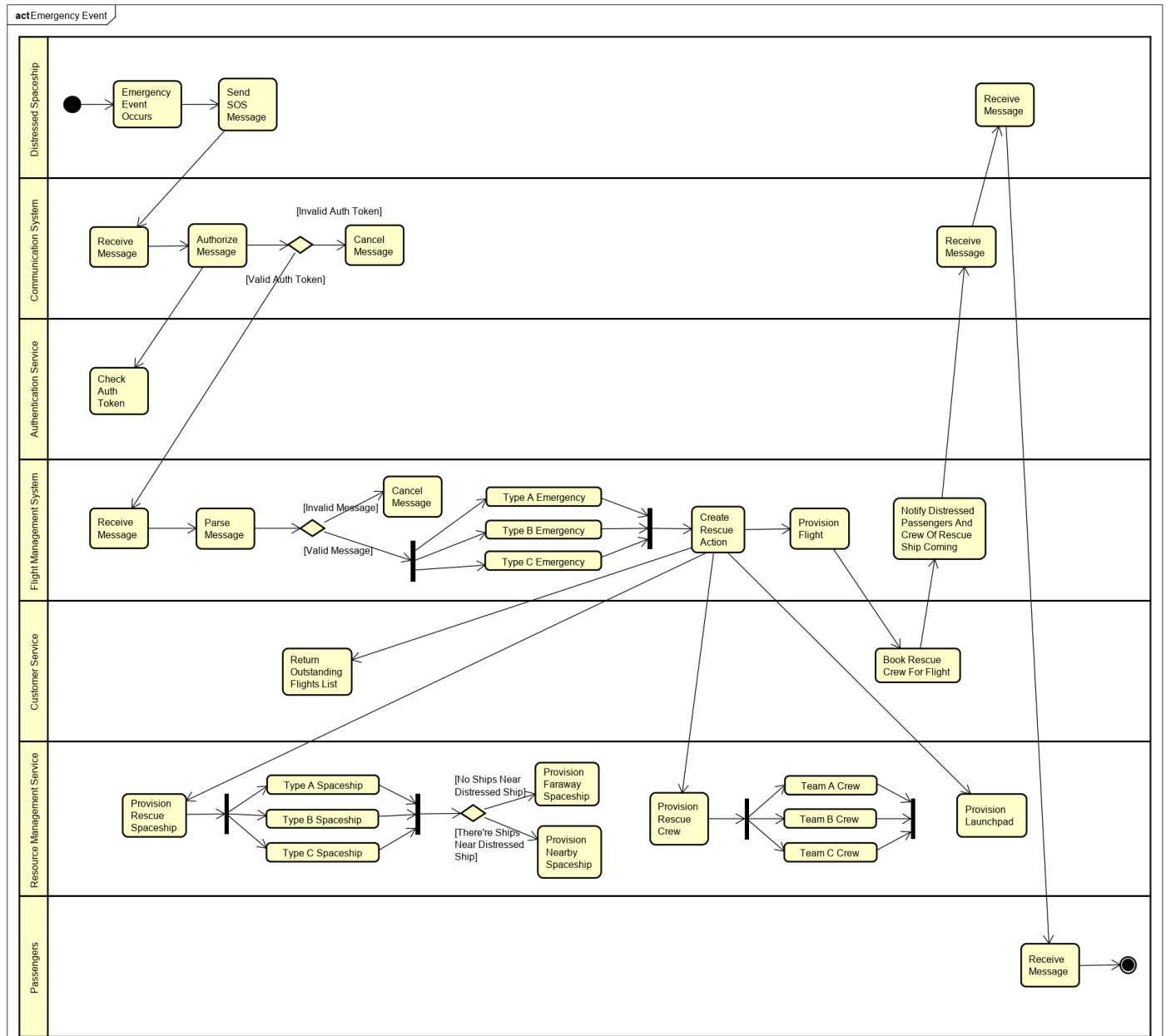
The ISTS will use the InterPlanetary File System for sharing and storing documents long term. It is assumed that connectivity to the IPFS will not be an issue on Earth or anywhere in space.

Use Cases

The IST system should support allowing administrators to manage, monitor, and update resources, people/teams, and flights. Administrators are tasked with managing communication and information between the resources including flight and spacecraft status, messages from spacecraft, and ground-based communication and automated control systems.

Activity Diagram

The following is an activity diagram illustrating the flow of activity for when an Emergency event occurs and a rescue mission is sent to the distressed spaceship in response:



Caption: Showing the message flow for when a spaceship sends a distressed signal and an emergency response mission is provisioned.

Implementation Details

In order to achieve level 5 modularity maturity as required, each module is implemented in its own package as a microservice and any components external to a package must use its service API in order to access it. Additionally, this exclusive use of an API for access hides the underlying details of a module's implementation. This exemplifies the Façade design pattern. In addition, each module will provide a Singleton instance of its service since multiple instances would be unnecessary and costly. This exemplifies the Singleton design pattern.

Also, it should be noted that my implementation of the Authentication service used a class called AuthTokenTuple (which includes an AuthToken) and is included in this document. For this design document it can be thought of as just an AuthToken to avoid confusion. Though please feel free to refer to my Authentication service implementation if more clarification is needed.

ISTS Resource Management Service Design Specification

Introduction

This design specification explains how to implement the Resource Management service module of the IST system. The Resource Management service provisions and maintains the state of the domain objects of the ISTS that aren't customer service related. Domain resources include human resources like Person and Team, and physical resources such as Spaceship and Budget.

Overview

The Resource Management service provides an API for the GUI to utilize that allows ISTS administrators to interact with the ISTS resources. It supports querying resource state as well as updating their state. In addition to provisioning resources, it is able to create/simulate events including those emitted by spaceships, the communication system, and the computer system which includes sensors.

Please refer to the component diagram in the previous Architectural Overview section to see a high-level overview of how the ISTS components fit together.

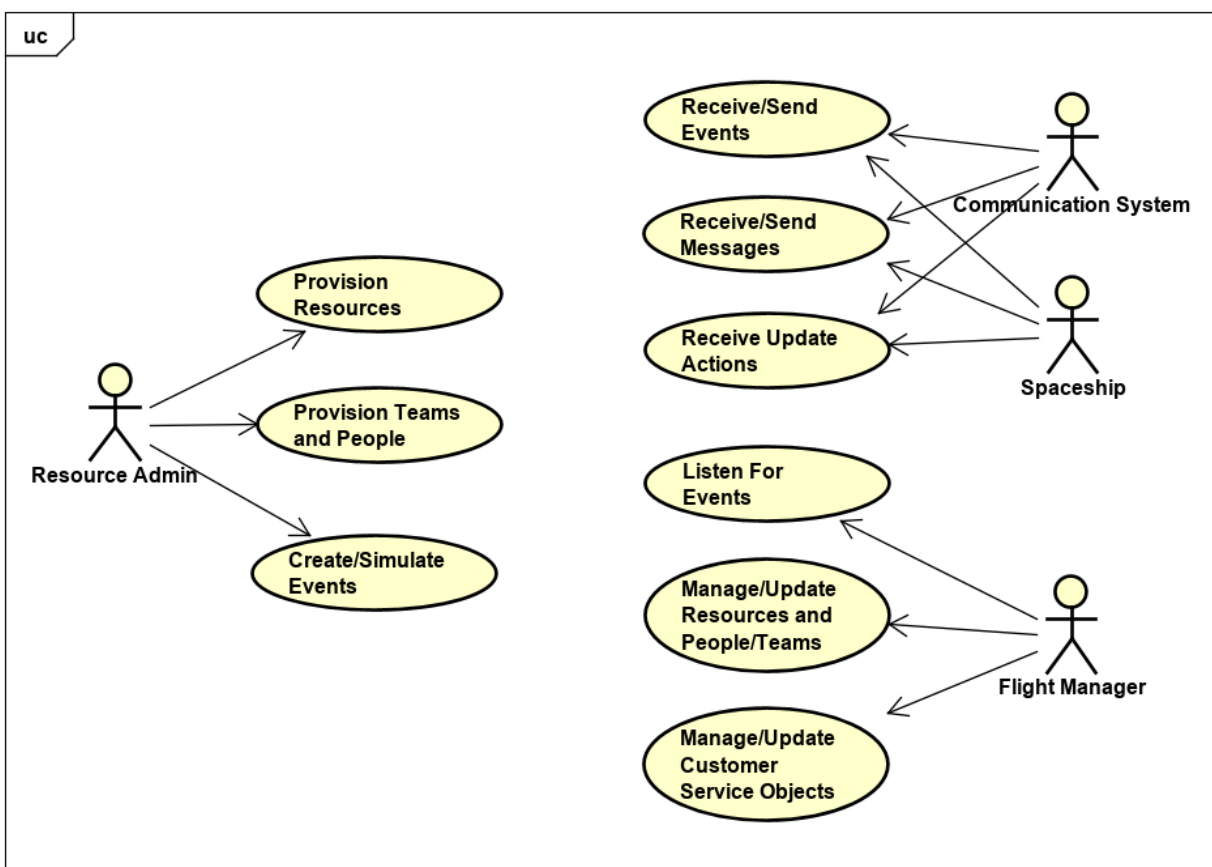
Requirements

The Resource Management System must be able to view as well as manage teams and people where people can be part of one or more teams. It must also manage the IST system's Ledger blockchain account since it has a resource budget that must be maintained to purchase resources with. All resource management including of physical assets like spaceships and launchpads need to support all four CRUD operations, i.e., create, read, update, and delete. Access to its domain objects' information is important because the GUI and other modules rely on them to function. For example, the Flight Management service might depend on the Resource service to find out a spaceship's capacity for flight creation and it will also be listening for interesting events in the Resource Management service in order to manage and update the ISTS. Other resources include fuel, communication system, and computer system.

The Resource service must also be able to create events including spaceship and communication system events and the spaceship and communication system must be able to send/receive messages. Spaceships must be able to receive update actions in response to events such as when a flight reaches its destination. Communication system and computer system status needs to also be monitored. For this module's design specification, events can be thought of as opaque strings. Finally, all API methods must accept an Auth Token parameter to support access control.

Use Cases

The diagram below illustrates the use cases of the Resource Management service.



Caption: The actors in the Resource Management service. Their use cases are pointed to and state what interaction each actor has with the system.

Actors:

The actors of the Resource Management service are Resource Admin, Communication System, Spaceship, and Flight Manager (or Manager for short).

Resource Admin

An Administrator is allowed to use any of the Resource service's methods. They can provision

the ISTS system and have full access to any of its resources.

Flight Manager

The Manager implements the Flight Management service. The Manager can be both an administrator or the automated control system that responds to events. It listens for interesting events emitted by the Resource Management service and responds with an appropriate action including requests to the Resource Service. For example, an event could be that a spaceship crashed which the Manager could respond to by deploying a spaceship from the Resource Management service for a rescue flight mission.

Communication System

The communication system serves as a mediating agent in the communication system of the ISTS. It is utilized in creating/simulating events and sending messages.

Spaceship

The spaceships are able to communicate update events about their status or the status of the system to the Resource Management service. This includes the use of sensors provided by the computer system, and messages.

Provision Resources

Administrators can provision the IST system with resources. Provisioning resources involves buying them first so an ISTS budget must be maintained.

Provision Teams and People

Administrators can provision and manage human resources such as the people and teams that make up the ISTS organization hierarchy.

Create/Simulate Events

An administrator can create/simulate spaceship and communication system events such as a spaceship getting lost in space.

Receive/Send Events

Spaceships and the communication system are able to send/receive simulated events that imitate real world events that could happen. Events should follow a specified syntax so that they can be translated.

Receive/Send Messages

Spaceships and the communication system should be able to receive and send messages per the requirements.

Receive Update Actions

Update actions should be able to be done to the spaceship by the automatic control system implemented by the Flight Management service in response to events. The status of the communication system should also be monitored and updateable per the requirements.

Listen For Events

The Flight Manager listens for interesting events that happen in the Resource Management service. It gets notified by the Resource Management service of such events when they occur.

Manage/Update Resources and People/Teams

The Manager executes update actions on Resource Management service objects in response to events.

Manage/Update Customer Service Objects

The Manager executes update actions on Customer service objects in response to events.

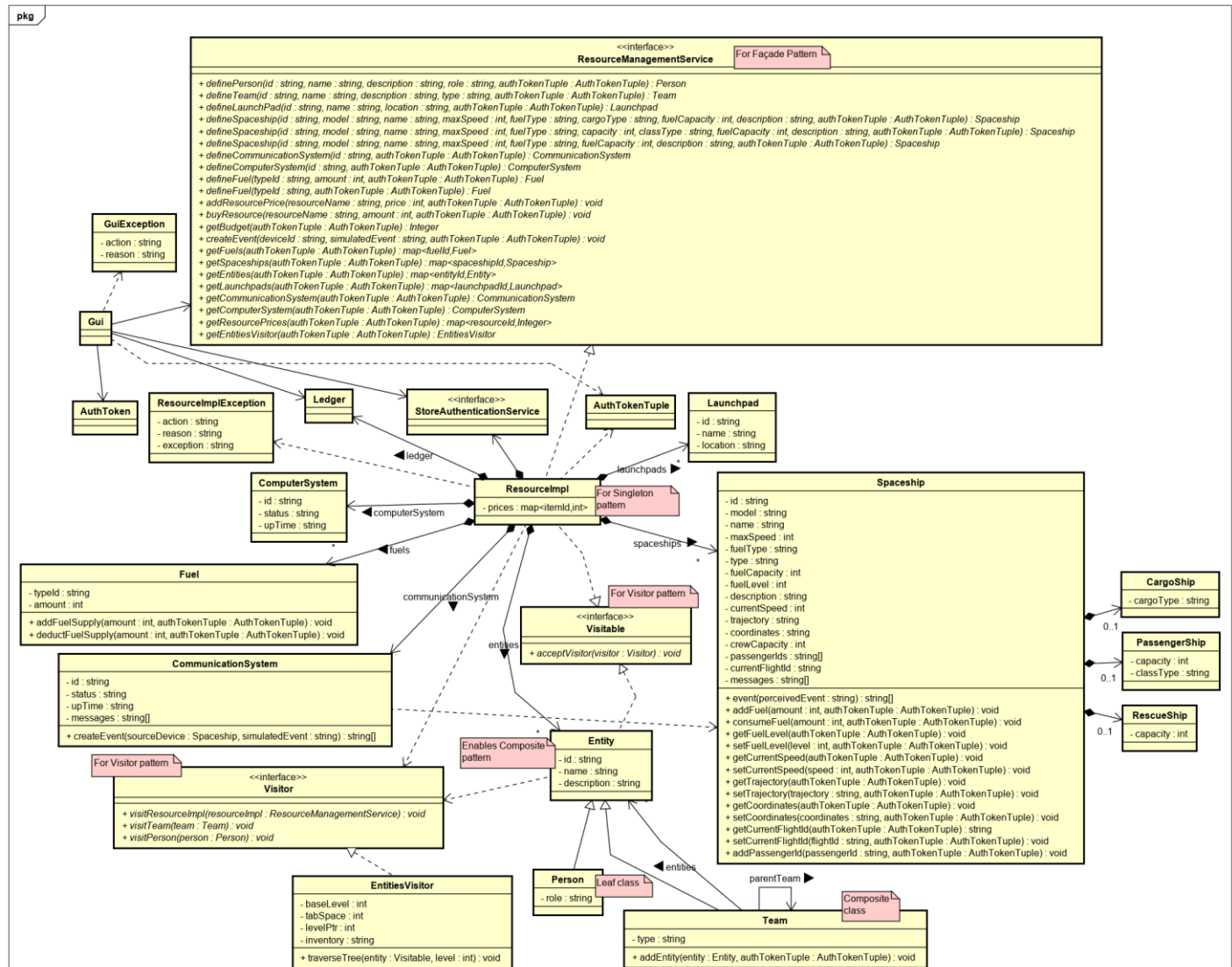
Implementation

This section explains how the Resource Management service will be implemented. Of note while reading this section is that the Resource Management service makes use of the Composite and Visitor design patterns with the human resources. This is expanded upon in the class diagram and class dictionary that follow.

It should also be noted that even though it's not convention to include, many accessors and mutators on the Spaceship were deliberately included in order to highlight the importance of their access permissions and their influence. Since there's a lot of accessing and updating that happens to the spaceships by the Flight Manager in response to update events, I wanted to bring attention to Spaceship getters and setters and also emphasize that they should have access permissions by showing them in my design and with AuthToken parameters. It's important that spaceships not be vulnerable to hacking.

Class Diagram

The following Resource Management service class diagram defines the implementation classes that comprise the package "com.cscie97.ists.resource" and includes classes from other packages that have an important relationship to the Resource Management service.



Caption: The class diagram for the Resource Management service. The pink notes indicate where the Composite, Visitor, Singleton, and Façade design patterns are being implemented.

Class Dictionary

This section contains the class dictionary for the Resource Management service that comprise the package “com.cscie97.ists.resource”.

ResourceManagementService (Interface)

The Resource Management service interface defines the API for the Resource Management service. Per the requirements, its exposed methods are the only point of access external components and entities can use for interacting with the service.

Methods

Method Name	Signature	Description
definePerson	(id : string, name : string, description : string, role : string, authTokenTuple : AuthTokenTuple) : Person	Creates a new Person within the IST organization with the given parameters and adds them to the map of entities.
defineTeam	(id : string, name : string, description : string, type : string, authTokenTuple : AuthTokenTuple) : Team	Creates a new Team within the IST organization with the given parameters and adds it to the map of entities.
defineLaunchPad	(id : string, name : string, location : string, authTokenTuple : AuthTokenTuple) : Launchpad	Creates a new Launchpad for spaceships with the given parameters and adds it to the map of launchpads.
defineSpaceship	(id : string, model : string, name : string, maxSpeed : int, fuelType : string, cargoType : string, fuelCapacity : int, description : string, authTokenTuple : AuthTokenTuple) : Spaceship	Creates a new Spaceship used for space flight with the given parameters and adds it to the map of spaceships. This is the method for cargo spaceships.
defineSpaceship	(id : string, model : string, name : string, maxSpeed : int, fuelType : string, capacity : int, classType : string, fuelCapacity : int, description : string, authTokenTuple : AuthTokenTuple) : Spaceship	Creates a new Spaceship used for space flight with the given parameters and adds it to the map of spaceships. This is the method for passenger spaceships.
defineSpaceship	(id : string, model : string, name : string, maxSpeed : int, fuelType : string, fuelCapacity : int, description : string, authTokenTuple : AuthTokenTuple) : Spaceship	Creates a new Spaceship used for space flight with the given parameters and adds it to the map of spaceships. This is the method for rescue spaceships.
defineCommunication System	(id : string, authTokenTuple : AuthTokenTuple) : CommunicationSystem	Creates a new CommunicationSystem used by the ISTS with the given id.
defineComputerSystem	(id : string, authTokenTuple : AuthTokenTuple) : ComputerSystem	Creates a new ComputerSystem used by the ISTS with the given id.
addResourcePrice	(resourceName : string, price : int, authTokenTuple : AuthTokenTuple) : void	Creates a new resource-to-price mapping for resource costs with the given parameters.

buyResource	(resourceName : string, amount : int, authTokenTuple : AuthTokenTuple) : void	Buys a resource using the Ledger service with the given parameters.
getBudget	(authTokenTuple : AuthTokenTuple) : Integer	Returns the current budget of the ISTS using its account balance in the Ledger service.
createEvent	(deviceId : string, simulatedEvent : string, authTokenTuple : AuthTokenTuple) : void	Creates the given simulated ISTS event as triggered by the device (e.g., spaceship) of the id given.
defineFuel	(typeId : string, amount : int, authTokenTuple : AuthTokenTuple) : Fuel	Creates a new Fuel with the given parameters. Types can be solar sail, ion drive, or oxygen/hydrogen. Amount parameter is the amount of fuel it should have.
defineFuel	(typeId : string, authTokenTuple : AuthTokenTuple) : Fuel	Method that creates a new Fuel but without an amount parameter. Types can be solar sail, ion drive, or oxygen/hydrogen.
getFuels	(authTokenTuple : AuthTokenTuple) : map<fuelId, Fuel>	Returns the Fuels.
getSpaceships	(authTokenTuple : AuthTokenTuple) : map<spaceshipId : Spaceship>	Returns the Spaceships.
getEntities	(authTokenTuple : AuthTokenTuple) : map<entityId : Entity>	Returns the Entities (Persons and Teams).
getLaunchpads	(authTokenTuple : AuthTokenTuple) : map<launchpadId : Launchpad>	Returns the Launchpads.
getCommunicationSystem	(authTokenTuple : AuthTokenTuple) : CommunicationSystem	Returns the CommunicationSystem.
getComputerSystem	(authTokenTuple : AuthTokenTuple) : ComputerSystem	Returns the ComputerSystem.
getResourcePrices	(authTokenTuple : AuthTokenTuple) :	Returns the resource-to-price mappings for the resource prices.

	map<resourceId, Integer>	
getEntitiesVisitor	(authTokenTuple : AuthTokenTuple) : EntitiesVisitor	Returns an EntitiesVisitor object which contains the current state of the entities in the ISTS in the form of an inventory string.

Visitable (Interface)

The Visitable interface for use in the Visitor design pattern. Objects that implement Visitable are typically one of many objects that are accessed (i.e., “visited”) in some sort of pattern in order to be read or updated for some purpose by an object that implements the Visitor interface (defined later in this specification).

Methods

Method Name	Signature	Description
acceptVisitor	(visitor : Visitor) : void	Calls the visit method on the given visitor that corresponds to a Visitable object passing in the object as a parameter.

ResourceImpl

The ResourceManagementService interface’s (discussed at the beginning of this dictionary) implementation Singleton class. Also implements the Visitable interface discussed above.

Properties

Property Name	Type	Description
prices	map<resourceId, Integer>	The resource-to-price mappings for the resource prices.

Associations

Association Name	Type	Description
entities	map<entityId, Entity>	The entities (Persons and Teams) that make up the ISTS organization hierarchy.
launchpads	map<launchpadId, Launchpad>	The launchpads that spaceships use for takeoff and landing.
spaceships	map<spaceshipId,	The spaceships used by the ISTS.

	Spaceship>	
communicationSystem	CommunicationSystem	The communication system used by the ISTS.
computerSystem	ComputerSystem	The computer system used by the ISTS including remote sensors spaceships and other devices use.
fuels	map<fuelId, Fuel>	The fuels that spaceships use.
ledger	Ledger	The Ledger service that must be used to buy resources and manage the ISTS budget with per the requirements.
authenticator	StoreAuthenticationService	The Authentication service that must be used for checking access permissions in every method per the requirements.

ResourceImplException

The ResourceImplException is thrown when errors occur in the ResourceImpl. It extends java.lang.Exception. The exception includes the action that was being performed and the reason for the exception.

Properties

Property Name	Type	Description
action	string	Command performed during exception occurrence.
reason	string	Reason for exception being thrown.
exception	string	Name of the exception being thrown, e.g., "ResourceImplException".

Spaceship

Represents the physical spaceship used in space flight travel in the ISTS. It can be one of three types of ships: Cargo, Passenger, and Rescue. It can send update events about its status through the Communication system and to the Resource service and send/receive messages. It can also be updated by external entities like the Flight Management service.

Methods

Method Name	Signature	Description
event	(perceivedEvent : string) : string[]	An event perceived or happening to the spaceship.
addFuel	(amount : int, authTokenTuple : AuthTokenTuple) : void	Adds fuel to the spaceship by the given amount.
consumeFuel	(amount : int, authTokenTuple : AuthTokenTuple) : void	Deducts the amount of fuel in the spaceship by the given amount.
getFuelLevel	(authTokenTuple : AuthTokenTuple) : void	Returns the fuel level of the spaceship.
setFuelLevel	(level : int, authTokenTuple : AuthTokenTuple) : void	Sets the fuel level of the spaceship by the given amount.
getCurrentSpeed	(authTokenTuple : AuthTokenTuple) : void	Returns the current speed of the spaceship.
setCurrentSpeed	(speed : int, authTokenTuple : AuthTokenTuple) : void	Sets the current speed of the spaceship by the given amount.
getTrajectory	(authTokenTuple : AuthTokenTuple) : void	Returns the trajectory of the spaceship.
setTrajectory	(trajectory : string, authTokenTuple : AuthTokenTuple) : void	Sets the trajectory of the spaceship with the given trajectory.
getCoordinates	(authTokenTuple : AuthTokenTuple) : void	Returns the coordinates of the spaceship.
setCoordinates	(coordinates : string, authTokenTuple : AuthTokenTuple) : void	Sets the coordinates of the spaceship to the given coordinates.
getCurrentFlightId	(authTokenTuple : AuthTokenTuple) : string	Returns the id of the flight the spaceship is currently associated with.
setCurrentFlightId	(flightId : string, authTokenTuple : AuthTokenTuple) : void	Sets the current flight id associated with the spaceship to the given id.
addPassengerId	(passengerId : string, authTokenTuple : AuthTokenTuple) : void	Adds the given passenger id to the list of IDs of passengers on the spaceship.

Properties

Property Name	Type	Description
id	string	The id of the spaceship.
model	string	The model of the spaceship.
name	string	The name of the spaceship.
maxSpeed	int	The maximum speed of the spaceship.
fuelType	string	Types can be solar sail, ion drive, or oxygen/hydrogen.
type	string	The type of the spaceship. Types can be cargo, passenger, or rescue.
fuelCapacity	int	The fuel capacity of the spaceship.
fuelLevel	int	The current fuel level of the spaceship.
description	string	A description of the spaceship.
currentSpeed	int	The current speed of the spaceship.
trajectory	string	The current trajectory of the spaceship.
coordinates	string	The current coordinates of the spaceship.
crewCapacity	int	The crew capacity of the spaceship.
passengerIds	string[]	The Passenger IDs of the passengers on the spaceship.
currentFlightId	string	The id of the flight the spaceship is currently associated with.
messages	string[]	The messages of the spaceship.

Associations

Association Name	Type	Description
passengerShip	PassengerShip	A nested class within the Spaceship class that contains important information if the spaceship is a passenger type.
cargoShip	CargoShip	A nested class within the Spaceship class that contains important information if the spaceship is a cargo type.

rescueShip	RescueShip	A nested class within the Spaceship class that contains important information if the spaceship is a rescue type.
------------	------------	--

Cargo

The Cargo class is nested in the Spaceship class and is instantiated for Spaceships that are of the cargo type. It holds extra and important information and behavior for these types of spaceships.

Properties

Property Name	Type	Description
cargoType	string	The type of cargo in a cargo spaceship. Types can be mining, satellite maintenance, construction equipment.

Passenger

The Passenger class is nested in the Spaceship class and is instantiated for Spaceships that are of the passenger type. It holds extra and important information and behavior for these types of spaceships.

Properties

Property Name	Type	Description
capacity	int	The capacity of a passenger spaceship.
classType	string	The class type of a passenger spaceship. Types can be luxury, or economy.

Rescue

The Rescue class is nested in the Spaceship class and is instantiated for Spaceships that are of the rescue type. It holds extra and important information and behavior for these types of spaceships.

Properties

Property Name	Type	Description
capacity	int	Passenger capacity for a rescue spaceship.

Entity

Entity is an abstract class that is amenable to the usage of the Composite design pattern. It also implements the Visitable interface so that it can be integrated into the Visitor design pattern.

Properties

Property Name	Type	Description
id	string	Unique id of the entity, e.g, "person 1", or "team 1".
name	string	The name of the entity.
description	string	Description of the entity.

Person

The Person class extends Entity and represents a person in the IST system and also a leaf object in the Composite pattern. As such, they can be nested within teams.

Properties

Property Name	Type	Description
role	string	The role of the person within the IST system.

Team

The Team class extends Entity and represents a group of people in the ISTS and also the composite in the Composite design pattern. As such, teams can be nested within other teams, and persons can be composed within teams.

Methods

Method Name	Signature	Description
addEntity	(entity : Entity, authTokenTuple : AuthTokenTuple) : void	Adds an entity to the team's nested entities.

Properties

Property Name	Type	Description
type	string	The team type. Types can be operations, flight crew, passenger, or rescue.

Associations

Association Name	Type	Description
entities	map<entityId, Entity>	The entities nested within the team.
parentTeam	Team	The team's parent team (if it has one).

Visitor (Interface)

Per the Visitor design pattern, the Visitor interface accesses (or “visits”) each of the Resource service entities in order to do something interesting with each such as read/write information or print information to stdout.

Methods

Method Name	Signature	Description
visitResourceImpl	(resourceImpl : ResourceManagementService) : void	Accesses the resourceImpl and does something interesting.
visitTeam	(team : Team) : void	Accesses a team and does something interesting.
visitPerson	(person : Person) : void	Accesses a person and does something interesting.

EntitiesVisitor

The EntitiesVisitor class implements the Visitor interface. It takes an inventory of the current entities (Persons and Teams) in the ISTS and collects interesting information in the process such as person roles and team names. This could be useful in scheduling crews or organizing passengers for flights (done by the Flight Management service) as it provides an organized view of the ISTS organization hierarchy, among other things.

Methods

Method Name	Signature	Description
-------------	-----------	-------------

traverseTree	(entity : Visitable, level : int) : void	Traverse resourceImpl's tree of entities to visit each entity and recursively on Team entities.
--------------	---	---

Properties

Property Name	Type	Description
baseLevel	int	The number of space indentations from the left margin for top-level entity objects in the entities tree (for readability when printed to stdout).
tabSpace	int	How many spaces are in one indentation.
levelPtr	int	A temporary pointer that tracks the levels in the entities structure.
inventory	string	The inventory of the entities (including any interesting information on them).

Launchpad

The Launchpad class represents where a spaceship will depart and arrive from.

Properties

Property Name	Type	Description
id	string	The id of the launchpad.
name	string	The name of the launchpad.
location	string	The location of the launchpad.

Fuel

Represents the fuel spaceships use and the supply of that fuel for the ISTS.

Methods

Method Name	Signature	Description
addFuelSupply	(amount : int, authTokenTuple : AuthTokenTuple) : void	Adds to the ISTS' resource supply of the fuel, e.g, when more fuel is purchased, by the given amount.

deductFuelSupply	(amount : int, authTokenTuple : AuthTokenTuple) : void	Deducts from the ISTS' resource supply of the fuel, e.g., when spaceships are filled with fuel, by the given amount.
------------------	--	--

Properties

Property Name	Type	Description
typeId	string	The type of the fuel. Types can be solar sail, ion drive, or oxygen/hydrogen.
amount	int	The amount of the fuel.

CommunicationSystem

This class represents an abstraction for large communication facilities. For the IST system to function, it's important that these facilities are online and available so their status must be monitored.

Methods

Method Name	Signature	Description
createEvent	(sourceDevice : Spaceship, simulatedEvent : string) : string[]	Used by the Resource Management service in the creation/simulation of events, e.g., spaceship events.

Properties

Property Name	Type	Description
id	string	The id of the communication system.
status	string	The status of the communication system. Status values can be up (online), or down (offline).
upTime	integer	How long the system has been up (online).
messages	string[]	The messages of the communication system.

ComputerSystem

This class represents an abstraction for large computer facilities including remote sensors. For the IST system to function, it's important that these facilities are up and available so their status

must be monitored.

Properties

Property Name	Type	Description
id	string	The id of the computer system.
status	string	The status of the system. Statuse values can be up (online), or down (offline).
upTime	Integer	How long the system has been up (online).

Gui

The Gui (Graphical User Interface) class represents the GUI that is used when interacting with the Resource Management service. The Gui class will be used to populate the UI and make it functional.

Associations

Association Name	Type	Description
resourceImpl	ResourceManagementService	The resourceImpl Singleton that implements the ResourceManagementService interface.
ledger	Ledger	The Ledger service for creating and managing accounts; the same one that the services use.
authenticator	StoreAuthenicationService	The Authentication service Singleton for logging into.
authToken	AuthToken	The Gui will need an AuthToken with the proper permissions to access services.

GuiException

The GuiException is thrown by errors in the Gui. It extends java.lang.Exception and includes the action that was being performed and the reason for the exception.

Properties

Property Name	Type	Description
action	string	Command performed when exception

		occurred.
reason	string	Reason for exception being thrown.

Implementation Details

The Resource Management service is largely an exercise in modeling. It's a very stateful service that provides and oversees the resources in the ISTS system from and on which other components can act. Per the requirements, objects need to be provisioned, accessible and updateable but not much more functionality is needed.

Nonetheless, one of the main functions of the ISTS is to provision and manage flights (done in the Flight Management service) using the Resource service. To be amenable to this end, Spaceships hold a list of passenger IDs which can be added to when passengers board the ship for their flight (Passenger is a Customer service class defined later in this document). In addition, the other Spaceship attributes of "capacity", "crewCapacity", and "currentFlightId" (Flight is a class in Customer service that is defined later in this document) should also prove useful as well as the EntitiesVisitor.

Exception Handling

To satisfy requirements and also for practicality, usability and good design, the following scenarios should be considered for exception handling:

- If the ISTS budget has insufficient funds for a transaction.
- If it is important to not have duplicate IDs in a namespace such as with Entity IDs.
- If attempting to fill past a spaceship's fuel or passenger capacity.
- When trying to use the computer or communication system when they're down.
- When objects are queried for that do not exist or can't be found.

An exception should be accompanied with useful information and named appropriately so that it's easy to understand why it was thrown.

Testing

Testing is done through a test driver class called TestDriver that contains a main method that accepts a command script file as a parameter. The script would have its own domain-specific language that corresponds to Resource service API method calls. The main method in TestDriver could exercise the Resource service API by reading in and parsing the script language and running the method calls. The TestDriver class should be set within the same classpath as the Resource Management service so that it can be run.

Risks

The in-memory implementation makes the system prone to losing the state of the resources. It is not required but the implementation could be updated to make use of the IPFS like the Customer service module does for long-term storage to mediate this.

Hackers may attempt to access the payment system. Transaction processing should be updated to require the payer account to sign transactions with a secure signature algorithm or similar.

The spaceship and resources don't have any functionality. Since events are opaque strings at this point, the system is inoperable and not self-sufficient. Functionality should be implemented to correct this.

Governmental regulations and standards on resources like spaceships, launchpads and fuel might conflict with the implementation or change unexpectedly especially considering that the ISTS is a new concept.

Abstractness and Instability Metrics

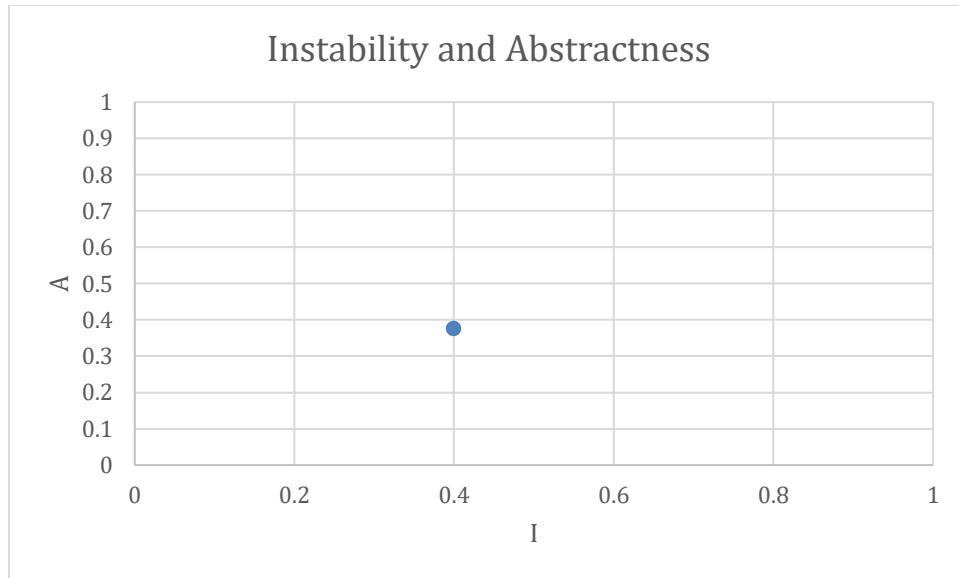
The afferent modules, C_a , for the Resource Management service are the Customer service, the Flight Management service, and the GUI. The efferent modules, C_e , are the Ledger service, and the Authentication service. This gives:

- $C_a = 3$ modules
- $C_e = 2$ modules
- $\text{Instability} = I = C_e / (C_e + C_a) = 2 / (2 + 3) = 0.4$

The concrete and non-instantiable classes, N_c , for the Resource Management service include the interfaces and abstract classes (Entity, Observer, ResourceManagementService, Subject, Visitable, and Visitor), N_a , plus the concrete classes (CommunicationSystem, ComputerSystem, EntitiesVisitor, Fuel, Launchpad, Person, ResourceImpl, Spaceship, Team, and UpdateEvent). This gives:

- $N_a = 6$
- $N_c = 16$
- $\text{Abstractness} = A = N_a / N_c = 6 / 16 = 0.375$

Following is the plot of Instability vs. Abstractness:



Caption: The plot of Instability vs. Abstractness metrics for the Resource Management Service.

In the chart above, the Resource Management service design seems to lean more toward the main sequence than anything else but also toward being stable. Its distance, D , from the main sequence:

$$D = |A + I - 1| = |0.375 + 0.4 - 1| = 0.225$$

ISTS Customer Service Design Specification

Introduction

This design specification explains how to implement the Customer service module of the IST system. The Customer service provisions and manages the domain objects of the ISTS that are customer related. This includes passenger interactions, travel documents, flight booking, customer feedback, in-flight entertainment, passenger registration, and points of interest.

Overview

The Customer service provides an API for the GUI to use that allows passengers and administrators to interact with Customer services. The Customer service focuses on things that support passenger needs and interactions with the ISTS. It also utilizes the InterPlanetary File System (IPFS) for sharing and storing files.

Please refer to the component diagram in the Architectural Overview section in the beginning of this document to see a high-level overview of how the ISTS components fit together.

Requirements

The Customer service must enable passengers (administrators can also be passengers) to register as participants in the ISTS system which involves submitting a Ledger service account and credentials (this can be pushed to the Authentication service for handling). Passengers could then login to their registered account with a credential in order to use the Customer service. Per the requirements, the Customer service should store and share its data on the InterPlanetary File System (IPFS). This includes travel documents (travel tickets, welcoming information packets, passports, and visas), in-flight entertainment (movies, music, and books), and discoveries and mission progress (includes text, and voice/video recordings). It is important that travel documents are accessible before a flight so that both passengers and flight staff can retrieve them for check-in and that customers are made aware of any important information. Passengers should not be allowed on a spaceship unless they have a ticket for the flight and the proper credentials. It is also important that in-flight entertainment be accessible to passengers to stream during the flight. When a flight has reached its destination, the system should ensure that discoveries and mission reports are saved to and accessible on the IPFS system.

For booking flights, passengers should have access to available destinations (or points of interest) as well as a list of available flights. Travel documents are created and a record of the booking is saved after a flight is booked. Passengers are also permitted to document their trip experience and provide feedback.

Booking passenger flights requires the use of the blockchain system for transactions, hence the need for passengers to submit their Ledger service accounts when registering. As with the Resource Management service, the Customer service should support all CRUD (create, read, update, and delete) operations in the management of its domain objects. Like the Resource Management service, the Customer service also receives update actions from the Flight Manager as part of the automated event control system and for flight provisioning. The Customer service should be able to use the create-event method and/or communication system in the Resource Management service to create events around its own domain objects such as when a new mission report is published that's interesting. Finally, all of its API methods must accept an Auth Token parameter to support access control.

Use Cases

The diagram on the next page illustrates the use cases of the Customer service.



Caption: The actors in the Customer service. Their use cases are pointed to and state what interaction each actor has with the system.

Actors:

The actors of the Customer service are Customer Admin, Passenger, and Flight Manager.

Customer Admin

A Customer Admin is permitted to use any of the Customer service methods. They can provision and have full access to Customer service objects. They can't however define flights as that is the job of the Flight Manager; they can only access and update parts of them (e.g., update passenger count after a flight booking). In addition to anything a passenger can do, Customer Admins can also manage points of interest, administer IPFS objects, and create events.

Passenger

A passenger can register as a participant in the IPFS and log in and log out of the Customer service portal where they can access Customer services. They should have access to points of interest offered by the ISTS and their accompanying flight information for booking flights. They also should have access to necessary travel documents needed before boarding a flight. During a flight they should be able to stream in-flight entertainment. They should also be able to document their experience and provide feedback.

Flight Manager

The Manager should be able to provision flights. The Manager can be both an administrator or the automated control system that responds to events. In response to a Discovery event, it may optionally be permitted to update any associated point of interest. Also, in response to a Reached Destination flight event, it needs to save discoveries and mission reports to the IPFS per the requirements.

Register Account

Passengers should be able to register as participants in the IST system. This includes submitting their blockchain account and adding credentials that identify them to the ISTS.

Login

Passengers should be able to use a username and password to login to the Customer service portal GUI. They should also be able to be recognized by the ISTS from a voiceprint and faceprint credential. The Authentication service should be utilized for this.

Logout

Passengers should be able to logout of the ISTS system. The Authentication service can be utilized for this.

Manage Points of Interest

The Customer Admin should be able to define and manage the points of interest that flights can travel to and from. The Flight Manager can optionally update a point of interest when, say, a new Discovery event happens there by adding the new discovery information to the point of interest as part of an automatic response update action.

Provision Flights

The Flight Manager is responsible for provisioning the flights of the ISTS. The Manager can be

both an administrator or the automated control system that responds to events.

Access Flight Information

Passengers need to be able to select from a list of available flights in order to book them. The Customer Admin must be able to update flight information. The Flight Manager must have access to the Customer service's flight list to provision flights.

Book Flights

Passengers should be able to book flights. Their Ledger accounts will be used in the transaction when purchasing a ticket.

Access Travel Documents

Passengers and flight staff should be able to access travel documents during check-in and flight boarding. Travel documents also include helpful information for flights.

Access In-Flight Entertainment

During the flight, passengers can access movies, music, and books for entertainment.

Record Discoveries / Mission Reports

Passengers should be able to record discoveries and mission reports they made on a trip. For example, a researcher passenger could record a newly discovered life form they found on a waypoint on their trip.

Document Experience

Passengers should be able to document their experience generally, e.g., for leisure.

Provide Feedback

Passengers should be able to provide feedback on their trip.

Administer IPFS Objects

As per the requirements, the Customer Admin is responsible for creating, reading, updating, and deleting any object that needs to be stored on the IPFS and managing their movement to and from the IPFS. Also per the requirements, the Flight Manager needs to save discoveries and mission reports to the IPFS in response to a Reached Destination flight event.

Create Events

The Customer service should also be able to create events so it must have access to the Resource Management service's method that creates events. Events involving the Customer service include the Mission Report, Discovery, and Customer Feedback events.

Implementation

This section explains how the Customer service will be implemented. Of note while reading this section is that the Customer service makes use of the Factory design pattern when moving

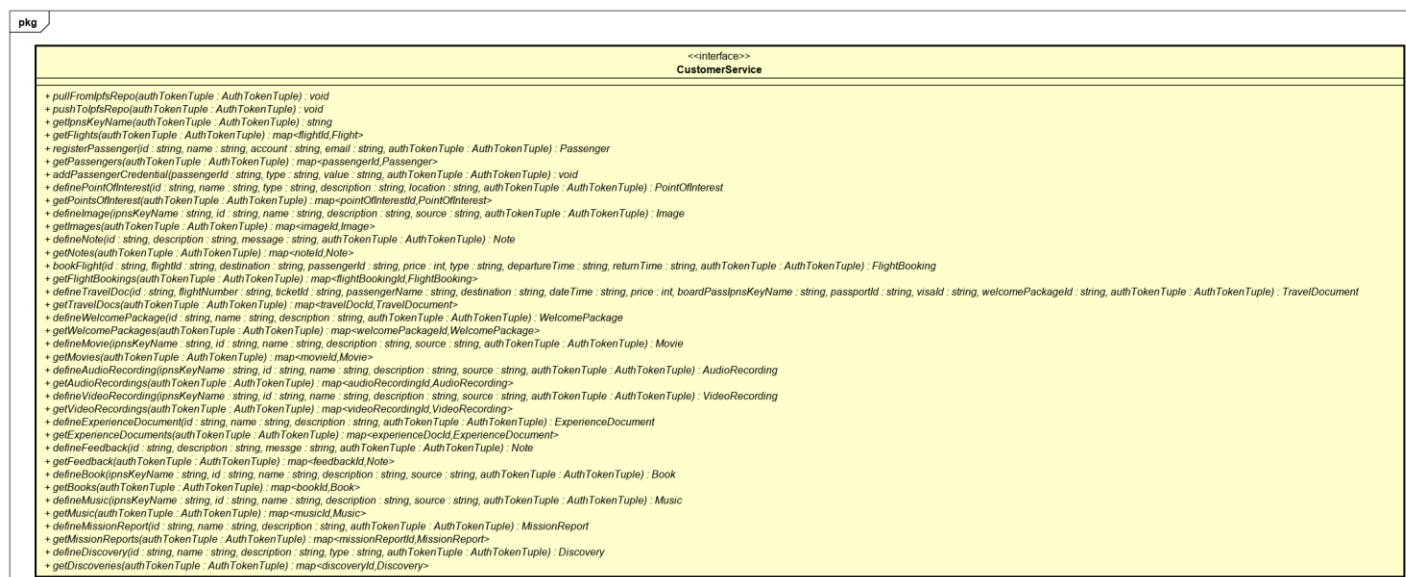
objects to and from the IPFS. This is expanded upon in the class diagram and class dictionary that follow. Also of note is that the IPFS has a key system for creating and updating mutable links to IPFS content called Inter-Planetary Name System (IPNS) which this design uses.

<https://docs.ipfs.io/guides/concepts/ipns/> This is expanded upon in the “Implementation Details” section later in this specification.

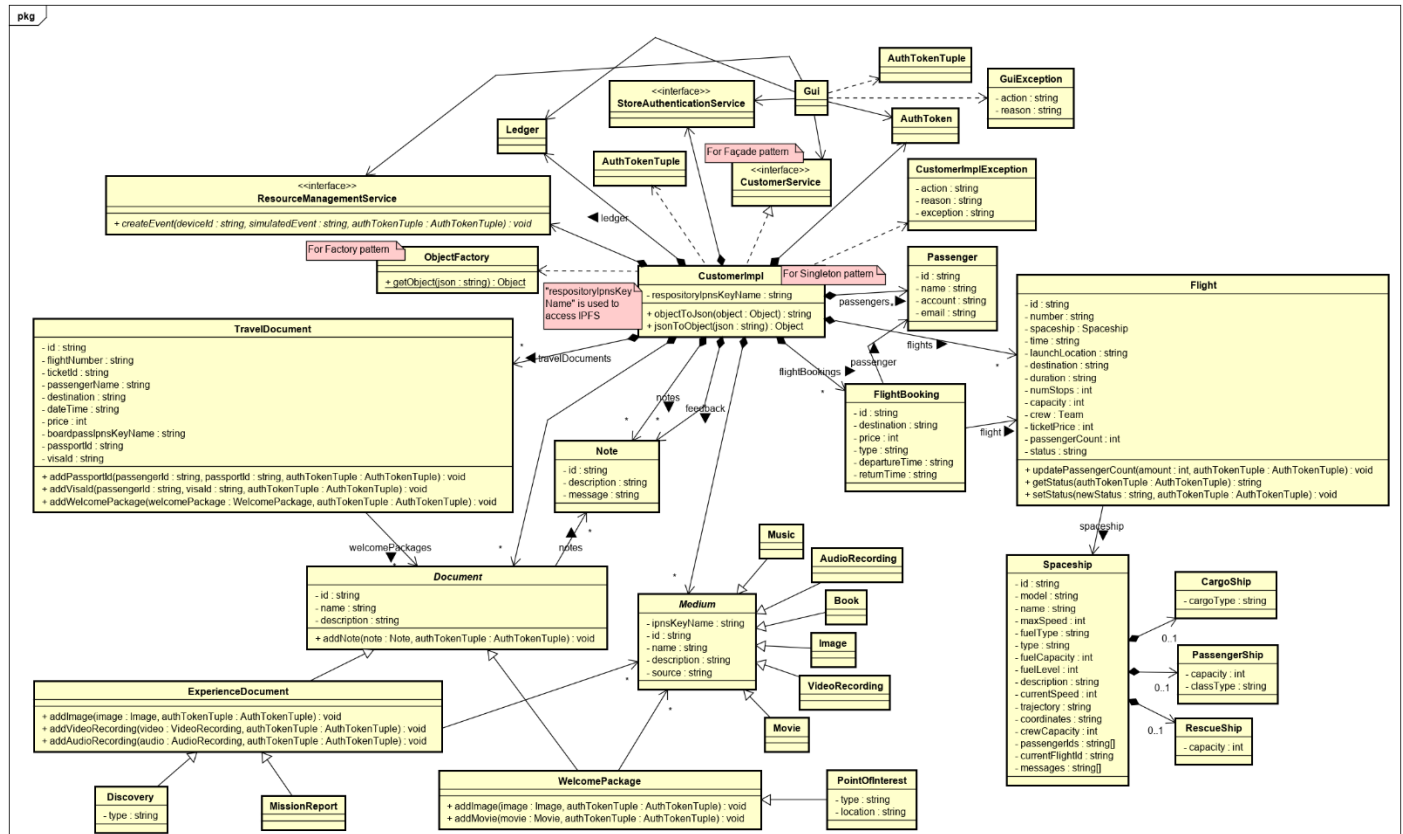
It should also be noted that even though it's not convention to include, many getters and setters were deliberately included in the Customer service API because there's a lot of privacy risks surrounding its function since it must handle a lot of data from many different users. I wanted to emphasize that not only are the Customer service's getters and setters important but they should have Auth Token parameters in order to apportion data with less privacy risk. This is expanded upon in the “Implementation Details” section later in this specification.

Class Diagram

The following Customer service API and class diagram defines the implementation classes that comprise the package “com.cscie97.ists.customer” and includes classes from other packages that have an important relationship to the Customer service. The CustomerService interface prefaces the class diagram with its complete list of methods in order to save space on the page and for legibility. In the class diagram that follows it, the methods of the interface are left out.



Caption: Interface for the Customer service; the API that is implemented and exposed publicly.



Caption: The class diagram for the Customer service. It should be noted that only information that's pertinent to showing the core functionality of the Customer service is shown. For example, not all the methods in the Resource Management service interface in the diagram are included. Please refer to the previous module specification in this document to review information that may be missing from the diagram. The pink notes indicate where design patterns are implemented including the Factory pattern, and important information.

Class Dictionary

This section contains the class dictionary for the Customer service that comprise the package “com.cscie97.ists.customer”.

CustomerService (Interface)

The Customer service interface defines the API for the Customer service. Per the requirements, its exposed methods are the only interaction point for external components and entities to use for access.

Methods

Method Name	Signature	Description
pullFromIpfRepo	(authTokenTuple :	Fetches and downloads all

	AuthTokenTuple) : void	the persisted Customer service data (stored as JSON) from the remote IPFS document database to local memory while merging any remote changes with local changes in the process. A utility class is used for this called ObjectFactory (defined later in this class dictionary) which implements the Factory design pattern for turning JSON strings into Customer service domain objects.
pushToIpfsRepo	(authTokenTuple : AuthTokenTuple) : void	Uploads all Customer service data in local memory to the remote IPFS JSON formatted document database for persistence while merging any local changes with remotes changes in the process. There are freely available libraries on the web for doing this that can convert Java objects to JSON strings, e.g., Jackson API.
getIpnsKeyName	(authTokenTuple : AuthTokenTuple) : string	Returns the IPNS key name that links to the remote IPFS Customer service document database.
getFlights	(authTokenTuple : AuthTokenTuple) : map<flightId, Flight>	Returns the map of flights.
registerPassenger	(id : string, name : string, account : string, email : string, authTokenTuple : AuthTokenTuple) : Passenger	Registers/creates a new Passenger account with the given parameters and adds it to the map of passengers. The “account” parameter is their Ledger service account.
getPassengers	(authTokenTuple : AuthTokenTuple) : map<passengerId, Passenger>	Returns the map of registered passengers.
addPassengerCredential	(passengerId : string, type : string, value : string,	Creates a new credential for the passenger of the id given

	authTokenTuple : AuthTokenTuple) : void	with the given parameters and adds it to the passenger's map of credentials on the Authentication Service.
definePointOfInterest	(id : string, name : string, type : string, description : string, location : string, authTokenTuple : AuthTokenTuple) : PointOfInterest	Defines a new PointOfInterest with the given parameters and adds it to the map of points of interest.
getPointsOfInterest	(authTokenTuple : AuthTokenTuple) : map<pointOfInterestId, PointOfInterest>	Returns the map of points of interest.
defineImage	(ipnsKeyName : string, id : string, name : string, description : string, source : string, authTokenTuple : AuthTokenTuple) : Image	Defines a new Image with the given parameters.
getImages	(authTokenTuple : AuthTokenTuple) : map<imageId, Image>	Returns the map of images.
defineNote	(id : string, description : string, message : string, authTokenTuple : AuthTokenTuple) : Note	Defines a new Note with the given parameters and adds it to the map of notes. The note's contents (written text) are contained in the "message" parameter.
getNotes	(authTokenTuple : AuthTokenTuple) : map<noteId, Note>	Returns the map of notes.
bookFlight	(id : string, flightId : string, destination : string, passengerId : string, price : int, type : string, departureTime : string, returnTime : string, authTokenTuple : AuthTokenTuple) : FlightBooking	Creates a new FlightBooking with the given parameters. Please see sequence diagram in the Implementation Details section later in this design specification to see the full process of booking a flight.
getFlightBookings	(authTokenTuple : AuthTokenTuple) : map<flightBookingId, FlightBooking>	Returns the map of flight bookings.

defineTravelDoc	(id : string, flightNumber : string, ticketId : string, passengerName : string, destination : string, dateTime : string, price : int, boardPassIpnsKeyName : string, passportId : string, visalId : string, welcomePackageld : string, authTokenTuple : AuthTokenTuple) : TravelDocument	Defines a new Travel Document with the given parameters and adds it to the map of travel documents. "boardPassIpnsKeyName" is a link to the passenger's boarding pass image stored on the IPFS.
getTravelDocs	(authTokenTuple : AuthTokenTuple) : map<travelDocId, TravelDocument>	Returns the map of travel documents.
defineWelcomePackage	(id : string, name : string, description : string, authTokenTuple : AuthTokenTuple) : WelcomePackage	Defines a new Welcome Package with the given parameters and adds it to the map of Welcome Packages.
getWelcomePackages	(authTokenTuple : AuthTokenTuple) : map<welcomePackageld, WelcomePackage>	Returns the map of welcome packages.
defineMovie	(ipnsKeyName : string, id : string, name : string, description : string, source : string, authTokenTuple : AuthTokenTuple) : Movie	Defines a new Movie with the given parameters and adds it the the map of movies.
getMovies	(authTokenTuple : AuthTokenTuple) : map<movieId, Movie>	Returns the map of movies.
defineAudioRecording	(ipnsKeyName : string, id : string, name : string, description : string, source : string, authTokenTuple : AuthTokenTuple) : AudioRecording	Defines a new AudioRecording with the given parameters and adds it to the map of audio recordings.
getAudioRecordings	(authTokenTuple : AuthTokenTuple) : map<audioRecordingId, AudioRecording>	Returns the map of audio recordings.

defineVideoRecording	(ipnsKeyName : string, id : string, name : string, description : string, source : string, authTokenTuple : AuthTokenTuple) : VideoRecording	Defines a new VideoRecording with the given parameters and adds it to the map of video recordings.
getVideoRecordings	(authTokenTuple : AuthTokenTuple) : map<videoRecordingId, VideoRecording>	Returns the map of video recordings.
defineExperienceDocument	(id : string, name : string, description : string, authTokenTuple : AuthTokenTuple) : ExperienceDocument	Defines a new ExperienceDocument with the given parameters and adds it to the map of experience documents.
getExperienceDocuments	(authTokenTuple : AuthTokenTuple) : map<experienceDocumentId, ExperienceDocument>	Returns the map of experience documents.
defineFeedback	(id : string, description : string, message : string, authTokenTuple : AuthTokenTuple) : Note	Creates a new Feedback and adds it to the map of feedbacks.
getFeedback	(authTokenTuple : AuthTokenTuple) : map<feedbackId, Note>	Returns the map of feedbacks.
defineBook	(ipnsKeyName : string, id : string, name : string, description : string, source : string, authTokenTuple : AuthTokenTuple) : Book	Defines a new Book and adds it to the map of books.
getBooks	(authTokenTuple : AuthTokenTuple) : map<bookId, Book>	Returns the map of books.
defineMusic	(ipnsKeyName : string, id : string, name : string, description : string, source : string, authTokenTuple : AuthTokenTuple) : Music	Defines a new Music and adds it to the map of musics.
getMusic	(authTokenTuple : AuthTokenTuple) : map<musicId, Music>	Returns the map of musics.

defineMissionReport	(id : string, name : string, description : string, authTokenTuple : AuthTokenTuple) : MissionReport	Defines a new MissionReport and adds it to the map of mission reports.
getMissionReports	(authTokenTuple : AuthTokenTuple) : map<missionReportId, MissionReport>	Returns the map of mission reports.
defineDiscovery	(id : string, name : string, description : string, type : string, authTokenTuple : AuthTokenTuple) : Discovery	Defines a new Discovery and adds it to the map of discoveries.
getDiscoveries	(authTokenTuple : AuthTokenTuple) : map<discoveryId, Discovery>	Returns the map of discoveries.

CustomerImpl

The CustomerService interface's (discussed above) implementation Singleton class.

Methods

Method Name	Signature	Description
objectToJson	(object : Object) : string	Utility method that converts a given Customer service domain object to JSON string. Used in the "pushToIpfsRepo" Customer service API method.
jsonToObject	(json : string) : Object	Utility method that converts a JSON string to a Customer service domain object. Utilizes the ObjectFactory class (discussed later in this class dictionary). Used in the "pullFromIpfsRepo" Customer service API method.

Properties

Property Name	Type	Description
repositoryIpnsKeyName	string	The IPNS key name that links to where the remote IPFS Customer service document database is stored.

Associations

Association Name	Type	Description
ledger	Ledger	The Ledger service used for transactions during flight booking.
resourceImpl	ResourceManagementService	A reference to the Resource Management service Singleton implementation. Used when creating events such as a “Customer Feedback” event and others.
authenticator	StoreAuthenticationService	A reference to the Authentication service Singleton. Checks access permissions and returns Auth Tokens (login).
passengers	map<passengerId, Passenger>	The map of passengers.
pointsOfInterest	map<id, PointOfInterest>	The map of points of interest.
images	map<imageId, Image>	The map of images.
notes	map<noteId, Note>	The map of notes.
flightBookings	map<id, FlightBooking>	The map of flight bookings.
travelDocuments	map<id, TravelDocument>	The map of travel documents.
welcomePackages	map<id, WelcomePackage>	The map of welcome packages.
experienceDocuments	map<id, ExperienceDocument>	The map of experience documents.
movies	map<movieId, Movie>	The map of movies.
videoRecordings	map<id, VideoRecording>	The map of video recordings.
audioRecordings	map<id, AudioRecording>	The map of audio recordings.
feedback	map<feedbackId, Note>	The map of feedbacks.
music	map<musicId, Music>	The map of musics.
books	map<bookId, Book>	The map of books.
flights	map<flightId, Flight>	The map of flights.
missionReports	map<id, MissionReport>	The map of mission reports.

discoveries	map<id, Discovery>	The map of discoveries.
authToken	AuthToken	The customerImpl needs an AuthToken with the appropriate permissions to interface with other components such as when it may need to interact with the resourceImpl to create events.

CustomerImplException

The CustomerImplException is thrown when errors occur in the CustomerImpl. It extends java.lang.Exception. The exception includes the action that was being performed and the reason for the exception.

Properties

Property Name	Type	Description
action	string	Command performed during exception occurrence.
reason	string	Reason for exception being thrown.
exception	string	Name of the exception being thrown, e.g., "CustomerImplException".

Flight

The Flight class represents a flight or scheduled spaceship trip from one place to another within the IST system.

Methods

Method Name	Signature	Description
updatePassengerCount	(amount : int, authTokenTuple : AuthTokenTuple) : void	Increases the number of passengers scheduled for the flight by the amount given.
getStatus	(authTokenTuple : AuthTokenTuple) : string	Gets the status of the flight. Statuses include "preparing for launch", "in-flight", "reached destination", "lost:missing", and "lost:spacecraft fault".
setStatus	(status : string, authTokenTuple : AuthTokenTuple) : void	Sets the status to the given string.

Properties

Property Name	Type	Description
id	string	The id of the flight.
number	string	The flight number.
time	string	The time of the flight.
location	string	The location of the flight, i.e., launchpad.
destination	string	The destination of the flight, e.g., point of interest.
duration	string	The duration of the flight.
numStops	int	The number of stops from origin to destination, i.e., waypoints.
capacity	int	The passenger capacity of the flight.
ticketPrice	int	The ticket price of the flight.
passengerCount	int	The number of passengers currently booked for the flight.
status	string	The status of the flight. Statuses include "preparing for launch", "in-flight", "reached destination", "lost:mission", "lost:spacecraft fault".

Associations

Association Name	Type	Description
spaceship	Spaceship	The spaceship that will be used for the flight.
crewId	Team	The crew id of the flight, e.g., the Team id (from Resource Management service).

Passenger

The Passenger class represents someone who has registered an account with the IST system. These people have booked a flight or plan to book a flight with the ISTS. Administrators can also be passengers.

Properties

Property Name	Type	Description
id	string	The id of the passenger.
name	string	The name of the passenger.
account	string	The Ledger service account of the passenger.
email	string	The email address of the passenger.

FlightBooking

Per the requirements, a record of a booked flight must be saved and this class serves that purpose. It is created in response to a passenger reserving a ticket for a flight and contains information pertinent to that flight.

Properties

Property Name	Type	Description
id	string	The id of the flight booking record.
destination	string	The destination of the flight that was booked.
price	int	The price of the ticket for the flight.
type	string	The type of ticket reservation. Types include “one way”, “round trip”, and “guided tour”.
departureTime	string	The departure time for the flight.
returnTime	string	The return time for the flight.

Associations

Association Name	Type	Description
flight	Flight	The flight associated with the flight booking record.
passenger	Passenger	The passenger that booked the flight.

Medium

Medium refers to the singular of media which are outlets of communication, e.g., images, music, and movies that comprise data other than simple text. The Medium class is an abstract class that can be extended by a subclass to represent a type of medium, e.g., an image or movie.

Properties

Property Name	Type	Description
ipnsKeyName	string	The key that links to the medium's file location on the IPFS.
id	string	The id of the medium.
name	string	The name of the medium.
description	string	A description of the medium.
source	string	The source of the medium. Citing sources protects from copyright infringement, among other things.

Image

Extends Medium and represents an image., e.g., document image, one taken by a passenger.

Movie

Extends Medium and represents a movie, e.g., for entertainment, an informational/welcoming video.

Music

Extends Medium and represents music, e.g., for entertainment.

Book

Extends Medium and represents a book. e.g., for entertainment.

VideoRecording

Extends Medium and represents a video recording, e.g., one taken by a passenger.

AudioRecording

Extends Medium and represents an audio recording, e.g., one taken by a passenger.

Note

A Note represents written text. They can be used in things like documents, and points of interest.

Properties

Property Name	Type	Description
id	string	The id of the note.
description	string	A description of the note.
message	string	The written text that make up the note.

Document

An abstract class that can contain Notes.

Methods

Method Name	Signature	Description
addNote	(note : Note, authTokenTuple : AuthTokenTuple) : void	Adds a Note to the map of notes.

Properties

Property Name	Type	Description
id	string	The id of the document.
name	string	The name of the document.
description	string	A description of the document.

Associations

Association Name	Type	Description
notes	map<noteld, Note>	The map of the document's notes.

WelcomePackage

Represents the welcome package that's included in the travel documents sent to passengers after booking a flight which includes welcoming and traveling information. It extends Document and can contain images and movies in addition to notes.

Methods

Method Name	Signature	Description
addImage	(image : Image, authTokenTuple : AuthTokenTuple) : void	Adds an Image to the map of images.
addMovie	(movie : Movie, authTokenTuple : AuthTokenTuple) : void	Adds a movie to the map of movies.

Associations

Association Name	Type	Description
images	map<imageld, Image>	The map of images.
movies	map<movield, Movie>	The map of movies.

ExperienceDocument

Extends Document and represents the documentation of a passenger's experience. In addition to notes, it can contain images, video recordings, and audio recordings.

Methods

Method Name	Signature	Description
addImage	(image : Image, authTokenTuple : AuthTokenTuple) : void	Adds an Image to the map of images.

addVideoRecording	(videoRecording : VideoRecording, authTokenTuple : AuthTokenTuple) : void	Adds a VideoRecording to the map of video recordings.
addAudioRecording	(audioRecording : AudioRecording, authTokenTuple : AuthTokenTuple) : void	Adds an AudioRecording to the map of audio recordings.

Associations

Association Name	Type	Description
images	map<imageId, Image>	The map of images.
videoRecordings	map<id, VideoRecording>	The map of video recordings.
audioRecordings	map<id, AudioRecording>	The map of audio recordings.

MissionReport

Extends ExperienceDocument and represents a mission report that an administrative passenger or another qualified person can make.

Discovery

Extends ExperienceDocument and represents the documentation of a discovery made by a researcher or other passenger.

Properties

Property Name	Type	Description
type	string	The type of discovery made. Types include life, minerals, and object.

PointOfInterest

Extends WelcomePackage and represents a point of interest or destination that the ISTS can fly passengers to.

Properties

Property Name	Type	Description
type	string	The type of point of interest. Types include planet, moon, asteroid, solar system, and space station.
location	string	The location of the point of interest.

TravelDocument

Represents information and items that a passenger would or might need for their booked flight/trip such as check-in and boarding documents.

Methods

Method Name	Signature	Description
addPassportId	(passengerId : string, passportId : string, authTokenTuple : AuthTokenTuple) : void	Adds the given passport id for the passenger of the id given. Passengers and administrators should both have access to this method. For instance, a passenger's passport information might be unavailable at the time of travel document creation but they can upload later on.
addVisaId	(passengerId : string, passportId : string, authTokenTuple : AuthTokenTuple) : void	Adds the given visa id for the passenger of the id given. Passengers and administrators should both have access to this method. For instance, a passenger's visa information might be unavailable at the time of travel document creation but they can upload it later on.
addWelcomePackage	(welcomePackage : WelcomePackage, authTokenTuple : AuthTokenTuple) : void	Adds a welcome package to the map of welcome packages.

Properties

Property Name	Type	Description
id	string	The id of the travel document.
flightId	string	The flight id of the booked flight.
ticketId	string	The ticketId of the booked flight.

passengerName	string	The passenger's name.
destination	string	The flight's destination.
dateTime	string	The time of the flight's departure.
price	int	The price of the ticket for the flight.
boardPassIpnsKey Name	string	The IPNS key name that links to a picture file of the boarding pass stored on the IPFS.
passportId	string	Proof that indicates the nationality of the passenger.
visaId	string	Proof that indicates that the passenger can travel to the destination.

Associations

Association Name	Type	Description
welcomePackages	map<id, WelcomePackage>	The map of welcome packages.

ObjectFactory

Implements the Factory design pattern. The class has one public static method that accepts a Customer service object in JSON string format from the remote IPFS document database where Customer service data is persisted and returns the corresponding Customer service domain object in the local format for use in local memory. For instance, one JSON input string could return a Travel Document, and another could return a Movie. These are then placed in local memory in their appropriate maps in the Customer service. All data from the Customer service is converted to JSON when it's uploaded to the IPFS for persistence so this class will be very useful in the downloading of that data back to local memory especially since uploading to and downloading from the IPFS will be occurring often with many users of the system, etc.

Methods

Method Name	Signature	Description
getObject	(json : string) : Object	Returns a Customer service domain object from the given JSON string.

Gui

The Gui (Graphical User Interface) class represents the GUI that is used when interacting with the Customer service. The Gui class will be used to populate the UI and make it functional.

Associations

Association Name	Type	Description
customerImpl	CustomerService	The customerImpl Singleton that implements the CustomerService interface.
resourceImpl	ResourceManagementService	The resourceImpl Singleton that implements the ResourceManagementService interface.
ledger	Ledger	The Ledger service for creating and managing accounts; the same one that the services use.
authenticator	StoreAuthenticationService	The Authentication service Singleton for logging into.
authToken	AuthToken	The Gui will need an AuthToken with the proper permissions to access services.

GuiException

The GuiException is thrown by errors in the Gui. It extends java.lang.Exception and includes the action that was being performed and the reason for the exception.

Properties

Property Name	Type	Description
action	string	Command performed when exception occurred.
reason	string	Reason for exception being thrown.

Implementation Details

The Customer service handle's everything customer service related in the IST system. The key actor in this component is the passenger and how they interact with the IST system. It provides a flight booking system that's similar in function to currently available online ticketing services for airline flights as well as access to services for before, during, and after a flight.

Nonetheless, the Customer service is connected to other components in important ways. For instance, the list of Persons from the Resource Management service should mirror the Passengers that register with the Customer service for systemic consistency and maximal functionality. This is done by adding a passenger as a Person in the Resource service when they register with the Customer service. This is handled by the Flight Manager in its Passenger Registered event response. The Customer service also needs a reference to the Resource service because it can also create events (such as the Passenger Registered event) and so it needs to utilize the create-event method to notify observers (i.e., the Flight Manager) of interesting mission reports, or discoveries and such.

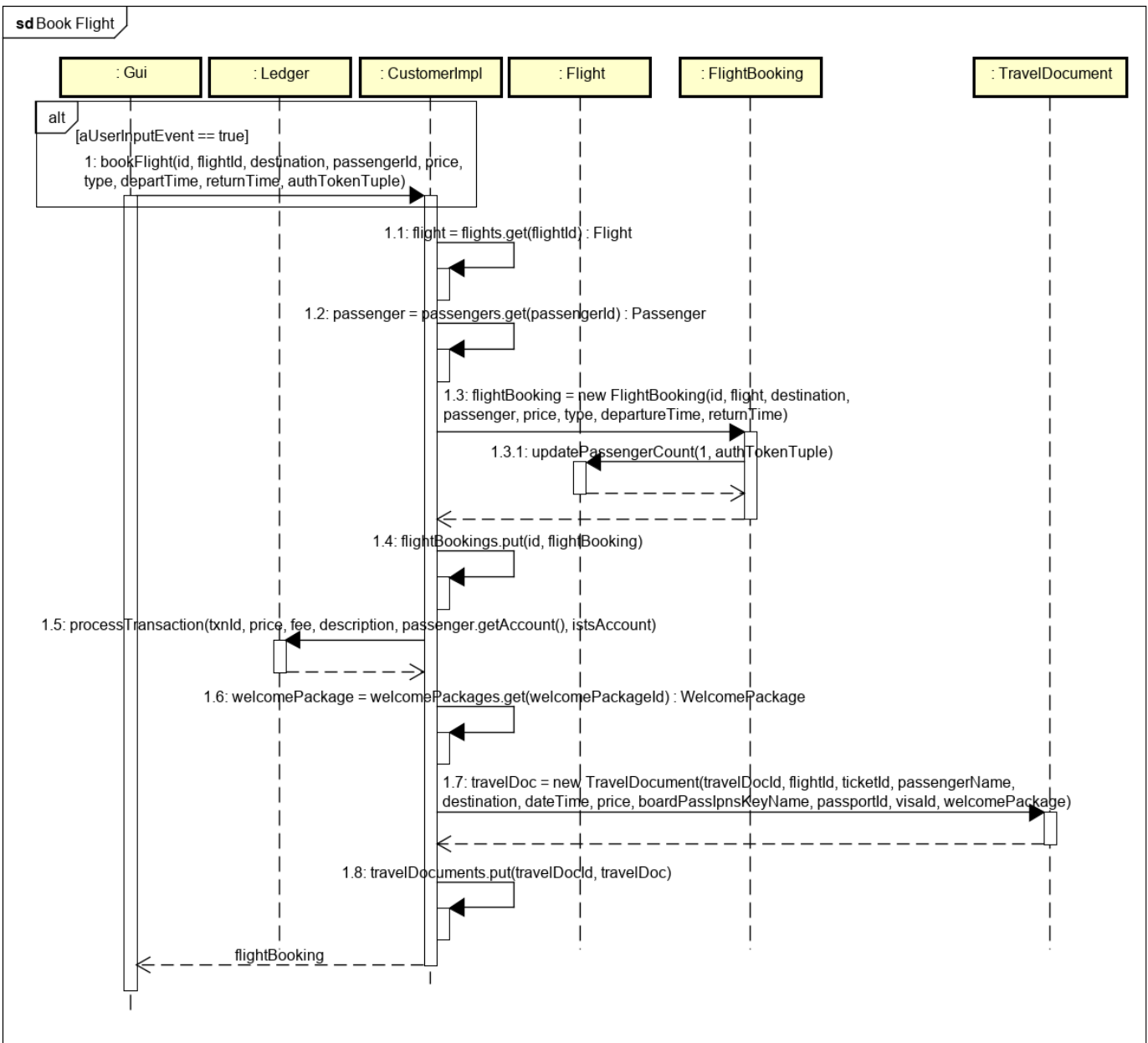
The Customer service also has a critical relationship with the IPFS component. It must persist Customer service objects by uploading them to the IPFS periodically, at critical times, and when directed. The IPFS has a key system for creating and updating mutable links to IPFS content called Inter-Planetary Name System (IPNS) which the Customer service uses. Typically, items stored on the IPFS are immutable – if the file changes, then the hash link to access the file necessarily changes. However, the IPNS system allows for mutable links that are human-readable, and easy to remember which is useful for this design because the document database on the IPFS that persists the Customer service data will be constantly accessed and updated. Administrators should be tasked with obtaining access to the IPNS key name that this design uses that links to the document database where all the Customer service data is persisted. This should be established and an IPFS database already set up by the time the ISTS becomes operational so that the “repositoryIpnsKeyName” attribute in the Customer service can be set on system startup via a bootstrap process – perhaps in the constructor of the Customer service – or some other method and the IPFS can be used.

Another important factor to consider with this design is that there’ll be a lot of data movement to and from the IPFS from many different users. Thus, there should be restrictions on accessors and mutators for privacy control. For instance, if a user calls the API’s “getVideoRecordings” method, they should only be able to access their own personal data store of video recordings and not another user’s (unless the user has administrative permissions in which case they can access anyone’s data). To have privacy control, the Customer service should use its methods’ AuthToken parameters to only return data that a user that’s calling a method has permissions to. And to facilitate this process, the Customer service should associate an object to a particular passenger by, for instance, systematically giving the domain objects IDs that include the passenger id in it. For example, a video recording given the id “passenger 1:videoRecording 1” would indicate that that particular video recording belongs to the passenger with id “passenger 1”.

The timing of when the Customer service moves data to and from the IPFS should also be carefully considered. For instance, flight bookings could be automatically uploaded to the IPFS (by calling the “pushToIpfsRepo” method) after they’re created as they are of relative importance. It would also be a good idea to automatically download the latest state of the remote IPFS data to a user’s local memory (by calling the “pullFromIpfsRepo” method) when

they sign in to the Customer service portal after a period of inactivity. Nonetheless, as a side note, a call to the “pushToIpfsRepo” method for persisting Customer service data should always be preceded by a call to the “pullFromIpfsRepo” method in order to synchronize local and remote data properly.

The following sequence diagram for booking a flight shows dependencies and interactions in the ISTS and how the Customer service is involved:



Caption: Sequence diagram showing the flow for booking a flight.

Exception Handling

To satisfy requirements and also for practicality, usability and good design, the following scenarios should be considered for exception handling:

- If trying to book more passengers on a flight than there is capacity for.
- If the passenger has insufficient funds for booking a flight.
- If there is a time conflict such that the user tries to book flights whose duration times overlap.
- When objects are queried for that do not exist or can't be found.
- When trying to create objects in the same namespace with duplicate IDs.

An exception should be accompanied with useful information and named appropriately so that it's easy to understand why it was thrown.

Testing

Testing is done through a test driver class called `TestDriver` that contains a main method that accepts a command script file as a parameter. The script would have its own domain-specific language that corresponds to Customer service API method calls. The main method in `TestDriver` could exercise the Customer service API by reading in and parsing the script language and running the method calls. The `TestDriver` class should be set within the same classpath as the Customer service so that it can be run.

Risks

The in-memory implementation makes the system prone to losing the state of the flights, passengers, flight bookings, and documents. It is not mentioned in the requirements but the implementation could be updated to make use of the IPFS for these objects like it does for the others for long-term storage.

The Customer service depends on the IPFS to function. There is overhead involved in establishing a working IPFS account and database. This should already be done and the IPFS account coupled to the Customer service before deployment for proper functioning.

The IPFS database that's being used also lacks security measures. To protect the integrity of the data, some data validation could be added to the design. For instance, data parameters could be validated before uploading to the IPFS.

Hackers may attempt to access the payment system. Transaction processing should be updated to require the payer account to sign transactions with a secure signature algorithm or similar.

Abstractness and Instability Metrics

The afferent modules, C_a , for the Customer service are the Flight Management Service, and the GUI. The efferent modules, C_e , are the Resource Management Service, Ledger Service, Authentication Service, and IPFS.

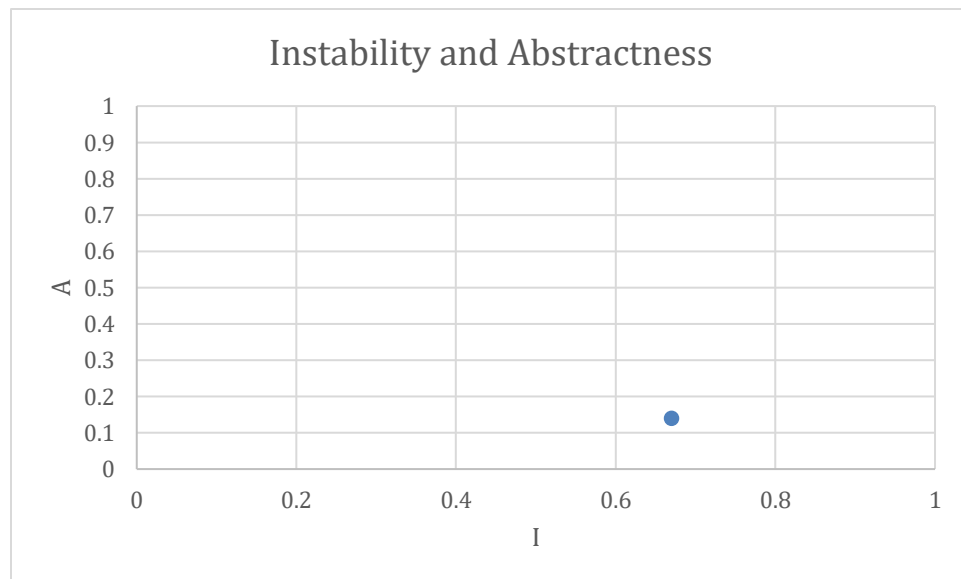
- $C_a = 2$ modules
- $C_e = 4$ modules
- $\text{Instability} = I = C_e / (C_e + C_a) = 4 / (4 + 2) = 0.67$

The concrete and non-instantiable classes, N_c , for the Customer service include the interfaces and abstract classes (CustomerService, Document, and Medium), N_a , plus the concrete classes (AudioRecording, Book, CustomerImpl, Discovery, ExperienceDocument, Flight, FlightBooking, Image, MissionReport, Movie, Music, Note, ObjectFactory, Passenger, PointOfInterest, TravelDocument, VideoRecording, and WelcomePackage). This gives:

- $N_a = 3$
- $N_c = 21$

$\text{Abstractness} = A = N_a / N_c = 3 / 21 = 0.14$

Below is the plot of Instability vs. Abstractness:



Caption: The plot of Instability vs. Abstractness metrics for the Customer service.

In the above chart, the design of the Customer service leans closer toward the main sequence than anything else but also towards being unstable and concrete. Its distance, D , from the main sequence:

$$D = |A + I - 1| = |0.14 + 0.67 - 1| = 0.19$$

ISTS Flight Management Service Design Specification

Introduction

This design specification explains how to implement the Flight Management service module of the IST system. The Flight Manager provisions, monitors, and manages flights. It also manages the “Space Command and Control System”, the automated event response system of the ISTS.

Overview

The Flight Management service provides an API for the GUI to utilize that allows ISTS administrators to provision flights. It also brings automated intelligence to the ISTS; it listens for interesting events occurring in the Resource Management service and has the ability to perform update actions in response to events throughout the IST system.

Please refer to the component diagram in the Architectural Overview section in the beginning of this document to see a high-level overview of how the ISTS components fit together.

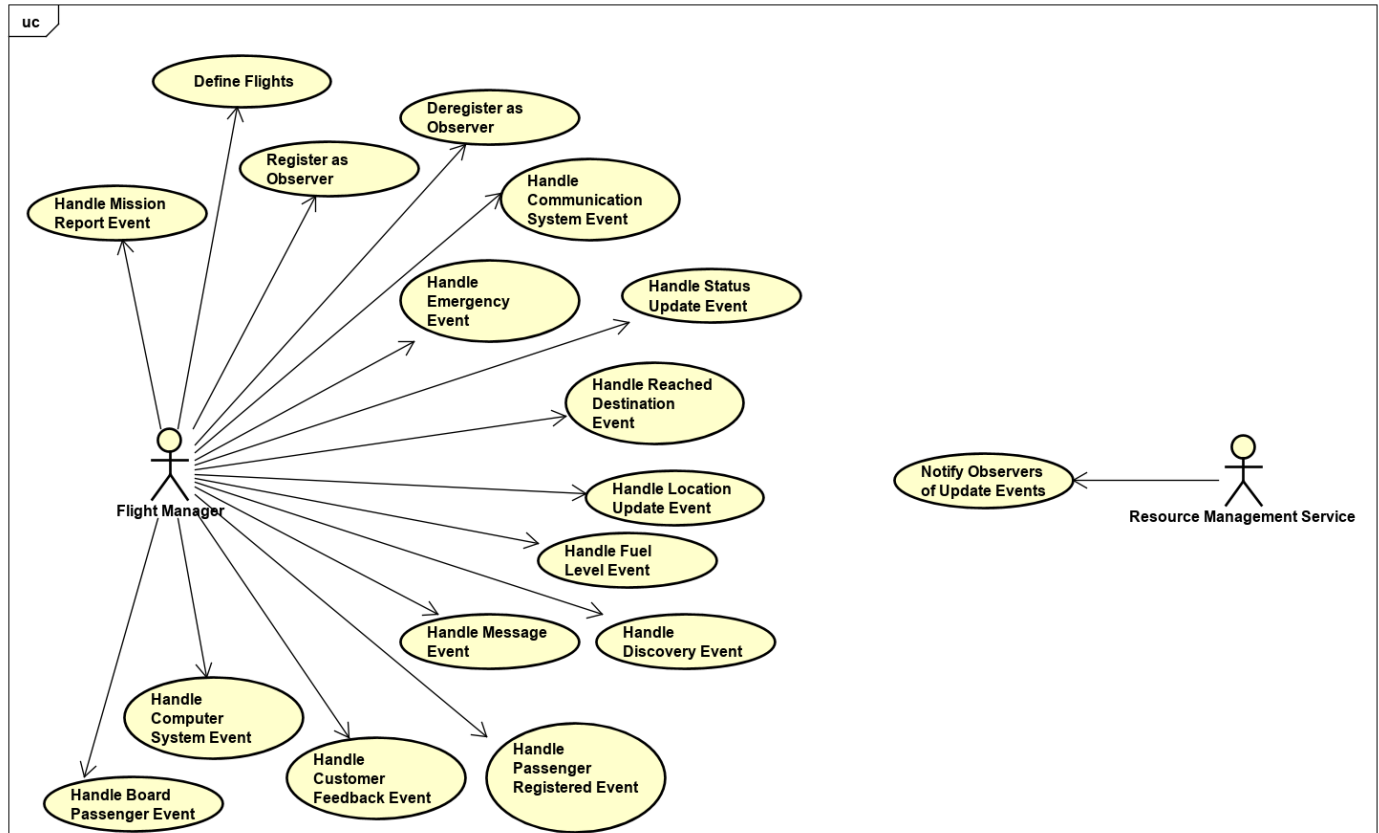
Requirements

In order to monitor and manage flights, the Flight Management service should implement the Observer interface defined by the Observer design pattern that allows it to be notified of interesting events by the Resource Management service where the ISTS communication/computer system and spaceships are accessible from. The Resource Management service in turn, should implement the Subject interface as defined by the Observer Pattern that would enable observers to register/deregister with it so that the Resource service can notify them of anything interesting. The Manager could then use the Command design pattern to create and execute rule-based update actions in response to update events. Events and resulting actions should be logged. The Manager is also responsible for provisioning flights for the ISTS so it must interface with the Resource service for accessing spaceships and human resources for flights including for rescue missions, and cargo flights.

The Manager is also tasked with updating some Customer service domain objects. For instance, when a flight reaches its destination, any newly created mission reports and discoveries need to be saved to and accessible on the IPFS. It may also, for instance, add information to a Customer service point of interest if a Discovery event occurs somewhere. In order to fulfill these duties, the Flight Manager needs to be able to interface with the Customer service where flights and other objects are accessible from. Finally, the method that defines flights must accept an Auth Token parameter to support access control.

Use Cases

The diagram below illustrates the use cases of the Flight Management service.



Caption: Shows the two main actors in the Flight Management service. Their use cases are pointed to and state what interaction each actor has with the system.

Actors:

The actors of the Flight Management service are Flight Manager and Resource Management service.

Flight Manager

The Manager listens to the Resource service for interesting events. When it observes an interesting event, it creates and executes the appropriate update actions in the ISTS based on a set of rules. For instance, if an Emergency event is observed, then it must create and execute a rescue flight mission to help the spaceship in distress that triggered the event. The Manager can be both an administrator or the automated control system that responds to events just discussed.

Resource Management Service

When an observer registers with the Resource Management service, the Resource service is

contractually obligated to notify it of any interesting update events. The Flight Manager needs to register with the Resource Management service to receive update notifications. The Resource service is able to create/simulate interesting events.

Notify Observers of Update Events

The Manager implements the Observer interface defined by the Observer design pattern which has an update method that the Resource service can use to notify it of update events. The Resource service must notify all of its observers of update events.

Register as Observer

Since the Resource service implements the Subject interface defined by the Observer design pattern, observers like the Flight Manager can use the Resource service's register method to register themselves with it.

Deregister as Observer

Since the Resource service implements the Subject interface defined by the Observer design pattern, observers like the Flight Manager can use the Resource service's deregister method to deregister themselves with it.

Handle Emergency Event

An Emergency event should trigger the Manager to do the "send rescue mission" action. This entails deploying a rescue flight mission to assist the spaceship in distress as soon as possible.

Handle Status Update Event

A Status Update event should trigger the Manager to do the "update flight status" action. This entails updating the status of the flight.

Handle Reached Destination Event

A Reached Destination event should trigger the Manager to do a "reached destination" action. This entails saving discoveries and mission reports to the IPFS, and updating the status of the flight.

Handle Location Update Event

A Location Update event should trigger the Manager to do a location-update action. This entails updating the speed, trajectory, and coordinates of the spaceship.

Handle Fuel Level Event

A Fuel Level event should trigger the Manager to do a "fuel level" action. This entails updating the spaceship's fuel level.

Handle Discovery Event

A Discovery event should trigger the Manager to do a "record discovery" action.

Handle Board Passenger Event

A Board Passenger event should trigger the Manager to do a “board passenger” action. This entails adding the passenger’s id to the spaceship’s list of passenger IDs.

Handle Mission Report Event

A Mission Report event should trigger the Manager to do an “analyze mission report” action.

Handle Customer Feedback Event

A Customer Feedback event should trigger the Manager to do a customer-feedback action.

Handle Passenger Registered Event

A Passenger Registered event should trigger the passenger-registered action. This entails adding the passenger that registered in the Customer service as a Person in the Resource service.

Handle Message Event

A Handle Message event should trigger the message action. This entails updating the messages of the sender and receiver.

Handle Communication System Event

A Communication System event should trigger the communication-system action. This entails updating the status of the communication system.

Handle Computer System Event

A Computer System event should trigger the computer-system action. This entails updating the status of the computer system.

Define Flights

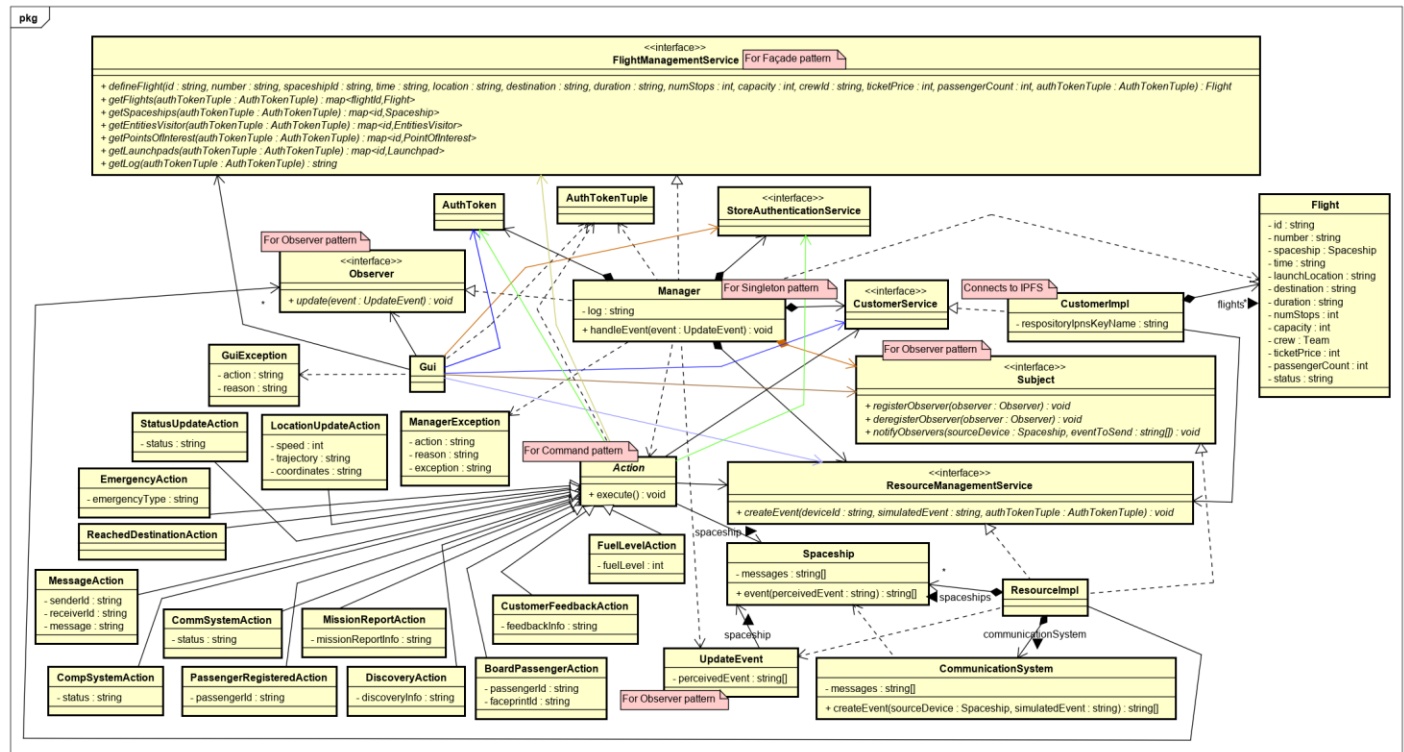
The Manager needs to provision all flights for the ISTS.

Implementation

This section explains how the Flight Management service will be implemented. Of note while reading this section is that the Flight Manager makes use of the Observer design pattern in its automated control system. Also of note is that the Manager’s automated control system uses the Command design pattern via its Action class in order to classify update events into actions. These are expanded upon in the class diagram and class dictionary that follow.

Class Diagram

The following Flight Management service class diagram defines the implementation classes that comprise the package “com.cscie97.ists.manage” and includes classes from other packages that have an important relationship to the Flight Management service.



Caption: Shows the class diagram for the Flight Management service. It should be noted that only information that is pertinent to showing the base coupling for how the Manager's automated control system manipulates the ISTS parts and information that is pertinent to how flights are provisioned is shown. For instance, not all of the Customer service's attributes, and methods are included in the diagram. Please refer to the previous two module specifications in this document to review information that may be missing from the diagram. The pink notes indicate where design patterns are being implemented, and important information.

Class Dictionary

This section specifies the class dictionary for the Flight Management service that comprise the package "com.cscie97.ists.manage" as well as classes from other packages that have an important relationship to the Flight Management service.

FlightManagementService (Interface)

The FlightManagementService interface provides the API for any external components to use for provisioning a flight.

Methods

Method Name	Signature	Description
defineFlight	(id : string, number : string, spaceshipId : string, time :	Defines a flight and adds it to the Customer service's map of flights.

	string, location : string, destination : string, duration : string, numStops : int, capacity : int, crewId : string, ticketPrice : int, passengerCount : int, authTokenTuple : AuthTokenTuple) : Flight	
getFlights	(authTokenTuple : AuthTokenTuple) : map<flightId, Flight>	Returns the map of flights from the Customer service. Useful for provisioning flights.
getSpaceships	(authTokenTuple : AuthTokenTuple) : map<id, Spaceship>	Returns the spaceships from the Resource Management service. Useful for provisioning flights.
getLaunchpads	(authTokenTuple : AuthTokenTuple) : map<id, Launchpad>	Returns the launchpads from the Resource Management service. Useful for provisioning flights.
getEntitiesVisitor	(authTokenTuple : AuthTokenTuple) : map<id, EntitiesVisitor>	Returns the EntitiesVisitor from the Resource Management service. Useful for provisioning flights.
getPointsOfInter est	(authTokenTuple : AuthTokenTuple) : map<id, PointOfInterest>	Returns the points of interest from the Customer service. Useful for provisioning flights.
getLog	(authTokenTuple : AuthTokenTuple) : string	Returns the log of events and their resulting actions.

Observer (Interface)

The Observer interface as defined by the Observer pattern has an update method that classes that implement the Subject interface use to notify the observer of an update event.

Methods

Method Name	Signature	Description
update	(event UpdateEvent): void	Receives an update event from a class that implements the Subject interface.

Subject (Interface)

The Subject interface as defined by the Observer pattern registers/deregisters observers, and

notifies all registered observers of an update event when it occurs.

Methods

Method Name	Signature	Description
registerObserver	(observer : Observer) : void	Adds a new observer to its list of observers to notify.
deregisterObserver	(observer : Observer) : void	Removes an observer from its list of observers to notify.
notifyObservers	(sourceDevice : Spaceship, eventToSend : string[]) : void	Notifies all observers of an update event.

Manager

Implements FlightManagementService and Observer interfaces. The Manager is aware of the Resource service and is able to register itself with it in order to receive update events.

Methods

Method Name	Signature	Description
handleEvent	(event : UpdateEvent) : void	A utility method that takes an update event as a parameter, maps it to an update action, and then executes the update action.

Properties

Property Name	Type	Description
log	string	A log of events and their resulting actions per the requirements.

Associations

Association Name	Type	Description
resourceImpl	ResourceManagementService; Subject	A reference to the Resource service so that the Manager can listen for events by registering itself with it.
customerImpl	CustomerService	Reference to the Customer service which the Manager performs update actions to and provisions flights for.

authenticator	StoreAuthenticationService	Reference to Authentication service for logging in / retrieving AuthTokens for itself as well as on behalf of passengers, e.g., to check if a passenger is permitted to board a plane using their faceprint. And also for checking access permissions on its method that defines flights.
authToken	AuthToken	The Manager needs an AuthToken with the proper access permissions to interface with the other services such as when defining flights for the Customer service.

ManagerException

The ManagerException is thrown when errors occur in the Manager. It extends java.lang.Exception. The exception includes the action that was being performed and the reason for the exception.

Properties

Property Name	Type	Description
action	string	Command performed during exception occurrence.
reason	string	Reason for exception being thrown.
exception	string	Name of the exception being thrown, e.g., "ManagerException".

ResourceImpl

The ResourceImpl implements the Subject interface defined by the Observer design pattern. It is only aware of the Flight Manager as an observer in this capacity and notifies it (and all its observers) whenever interesting events occur.

Methods

Method Name	Signature	Description
createEvent	(deviceId : string, simulatedEvent : string, authTokenTuple : AuthTokenTuple) : void	Sends the given event to the device, e.g., a spaceship, of the given id to be parsed and/or for processing.

Associations

Association Name	Type	Description
observers	Observer[]	The list of registered observers to notify when update events occur.
spaceships	map<spaceshipId, Spaceship>	The ISTS spaceships. They are able to trigger events such as the Reached Destination event.
communicationSystem	CommunicationSystem	The communication system of the ISTS that events and messages are communicated over.
authenticator	StoreAuthenticationService	The Authentication service. Used to check access permissions.

CommunicationSystem

Represents the communication system of the ISTS and is used by the services and devices for communication.

Methods

Method Name	Signature	Description
createEvent	(sourceDevice : Spaceship, simulatedEvent : string) : string[]	The communication system may be involved in created/simulated events and provides communication to devices like spaceships.
messages	string[]	The messages of the communication system.

Spaceship

Represents a spaceship in the ISTS. Spaceships are able to trigger events for the Resource Management to notify observers of.

Methods

Method Name	Signature	Description
event	(perceivedEvent : string) : string[]	The event that the spaceship perceived or happened to it.
messages	string[]	The messages of the spaceship.

UpdateEvent

The UpdateEvent class represents an event that is sent from a Subject (e.g., the Resource Management service) to an Observer (e.g., the Flight Manager) defined by the Observer pattern. It is created in the notifyObservers method of the Subject when it calls an Observer's update method.

Properties

Property Name	Type	Description
perceivedEvent	string[]	The event that the source device (e.g., spaceship) perceived and sent.

Associations

Property Name	Type	Description
sourceDevice	Spaceship	The device (e.g., spaceship) that sent the originating event, e.g., to the Resource Management service for notifying observers.

Action

The Action class encapsulates the actions that need to be performed in its execute method in response to an event including the update actions to a spaceship, and to the Resource Management and Customer services. It is an abstract class and per the Command design pattern that it implements, each event should map to an Action subclass based on classification rules, e.g., the type of event. When a new Action is instantiated, its one method – the execute method – is immediately called.

Methods

Method Name	Signature	Description
execute	() : void	Defines the executable action(s) that is needed in response to an event.

Associations

Property Name	Type	Description
sourceDevice	Spaceship	The source device (e.g., spaceship) that perceived/sent the original event and that

		update actions can be performed on if need be.
resourceImpl	ResourceManagementService	A reference to the Resource Management service that the self-sufficient Action classes may depend on in their execute methods, e.g., to schedule a spaceship for a flight.
customerImpl	CustomerService	A reference to the Customer service that the self-sufficient Action classes may depend on in their execute methods, e.g, to save mission reports to the IPFS.
manager	FlightManagementService	A reference to the Flight Management service that the self-sufficient Action classes may depend on in their execute methods, e.g, to provision a flight.
authToken	AuthToken	To perform their functions, the Action classes need an authToken with the appropriate access permissions.

EmergencyAction

The EmergencyAction class extends the Action class. It is created and executed automatically in response to an Emergency event per the requirements. Its execute method should provision a rescue flight mission to assist the spaceship in distress.

Properties

Property Name	Type	Description
emergencyType	string	The type of emergency.

StatusUpdateAction

Extends the Action class and is created and executed automatically in response to a Status Update event per the requirements. Its execute method should update the spaceship's status to the one given in the event.

Properties

Property Name	Type	Description
status	string	The status to change the spaceship's status to.

ReachedDestinationAction

Extends the Action class and is created and executed automatically in response to a Reached Destination event per the requirements. Its execute method should create and execute a Location Update action as well as save any published mission reports and discoveries to the IPFS by calling the Customer service's "pullFromIpfsRepo" and "pushToIpfsRepo" methods in succession which uploads all changes made on the local Customer service database to the remote IPFS database for persistence.

LocationUpdateAction

Extends the Action class and is created and executed automatically in response to a Location Update event. Its execute method updates the spaceship's speed, location, and trajectory.

Properties

Property Name	Type	Description
speed	int	The speed to update the spaceship to.
trajectory	string	The trajectory to update the spaceship to.
coordinates	string	The coordinates to update the spaceship to.

FuelLevelAction

Extends the Action class and is created and executed automatically in response to a Fuel Level event. Its execute method updates the fuel level of a spaceship.

Properties

Property Name	Type	Description
fuelLevel	int	The fuel level to update the spaceship to.

MissionReportAction

Extends the Action class and is created and executed automatically in response to a Mission Report event. Per requirements, its execute method must "analyze mission report".

Properties

Property Name	Type	Description
missionReportInfo	string	The mission report information needed to analyze it.

DiscoveryAction

Extends the Action class and is created and executed automatically in response to a Discovery event. Per the requirements, its execute method should “record discovery”.

Properties

Property Name	Type	Description
discoveryInfo	string	The discovery information needed to record it.

CustomerFeedbackAction

Extends the Action class and is created and executed automatically in response to a Customer Feedback event.

Properties

Property Name	Type	Description
feedbackInfo	string	The feedback information needed to act on it, e.g., record it.

PassengerRegisteredAction

Extends the Action class and is created and executed automatically in response to a Passenger Registered event. Its execute method should add the given passenger that registered with the Customer service as a Person in the Resource service.

Properties

Property Name	Type	Description
passengerId	string	The id of the passenger who should be added to the list of Persons in the Resource service.

BoardPassengerAction

Extends the Action class and is created and executed automatically in response to a Board Passenger event. Its execute method should authorize the given passenger boarding the spaceship for their flight using their faceprint credential. If the passenger is authorized, it should then add the passenger's id to the spaceship's list of passenger id's. Per the requirements, a "board flight" use case is required and this class is for meeting that requirement.

Properties

Property Name	Type	Description
passengerId	string	The passenger's id.
faceprintId	string	The faceprint of the passenger detected by a sensor.

MessageAction

Extends the Action class and is created and executed automatically in response to a Message Event. Its execute method should update the messages of the parties involved.

Properties

Property Name	Type	Description
senderId	string	The sending party of the message.
receiverId	string	The receiving party of the message.
message	string	The message being sent.

CommSystemAction

Extends the Action class and is created and executed automatically in response to a Communicate System event. Its execute method should change the communication system's status to the status given.

Properties

Property Name	Type	Description
status	string	The status to change the communication system to.

CompSystemAction

Extends the Action class and is created and executed automatically in response to a Computer System event. Its execute method should change the computer system's status to the status given.

Properties

Property Name	Type	Description
status	string	The status to change the computer system to.

Gui

The Gui (Graphical User Interface) class represents the GUI that is used when interacting with the Flight Management service. The Gui class will be used to populate the UI and make it functional.

Associations

Association Name	Type	Description
manager	FlightManagementService; Observer	The manager Singleton that implements the FlightManagementService.
customerImpl	CustomerService	The customerImpl Singleton that implements the CustomerService interface.
resourceImpl	ResourceManagementService; Subject	The resourceImpl Singleton that implements the ResourceManagementService interface.
ledger	Ledger	The Ledger service for creating and managing accounts; the same one that the services use.
authenticator	StoreAuthenticationService	The Authentication service Singleton for logging into.
authToken	AuthToken	The Gui will need an AuthToken with the proper permissions to access services.

GuiException

The GuiException is thrown by errors in the Gui. It extends java.lang.Exception and includes the action that was being performed and the reason for the exception.

Properties

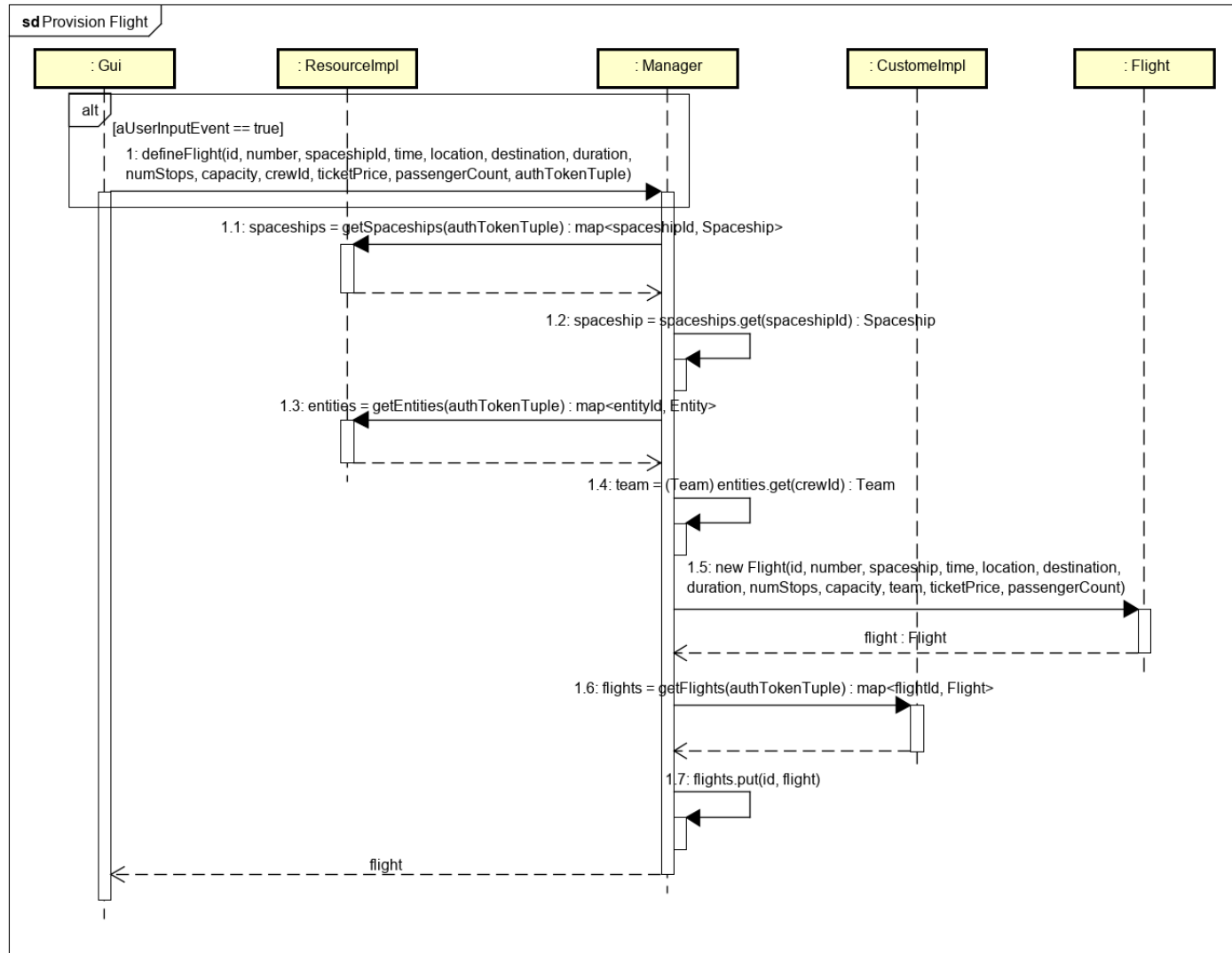
Property Name	Type	Description
action	string	Command performed when exception occurred.
reason	string	Reason for exception being thrown.

Implementation Details

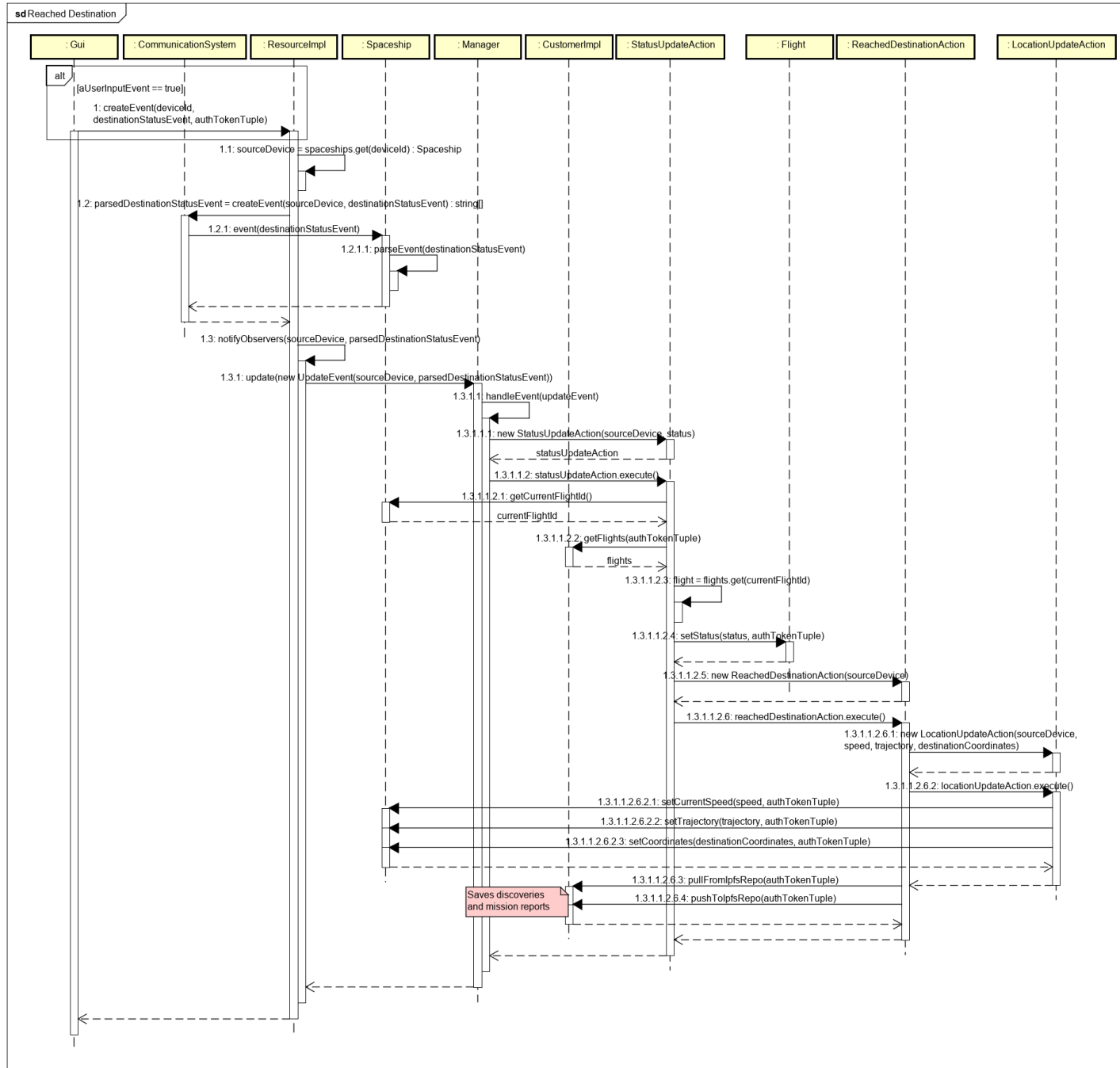
The Flight Management service is the brains of the ISTS. Its main class, Manager, registers with the Resource Management service as defined by the Observer design pattern in order to receive update events. When an event is observed, it must translate it, and then instantiate the appropriate Action class for execution as defined by the Command design pattern, e.g., some logic set is used to map events to actions. Uninteresting or unrecognized events can be ignored. The Flight Management service is also tasked with provisioning all flights.

Also, the underlying implementation of the Observer design pattern is such that the Flight Manager is aware of the Resource service but not vice versa. It needs to register with it after all. Thus, the scaffolding for the Observer design pattern – the Observer and Subject interfaces – should lie with the Resource service module. This also helps to avoid circular dependencies. One way to aware the Flight Manager of the Resource Management service is to include it as a parameter in the Manager's constructor.

As the processing center of the ISTS, the Flight Management service is a very behavior-oriented service. As such, it is commissioned with overseeing state in the other two main modules – the Resource Management service and the Customer service. Per requirements, the only interaction point for these services is through their interfaces. Consider, the sequence diagram below and the one that follows it for provisioning a flight and handling a Reached Destination event, respectively. Both show dependencies and interactions in the ISTS and how the Manager is involved:



Caption: Sequence diagram for showing the flow for how the Flight Management Service provisions a flight for the ISTS.



Caption: Sequence diagram for showing the flow for how the Flight Management service handles a Reached Destination event.

Exception Handling

To satisfy requirements and also for practicality, usability and good design, the following scenarios should be considered for exception handling:

- If an update event is not recognized, invalid, or not interesting.
- If a flight can't be schedule for whatever reason (e.g., time conflicts, crew availability,

etc.)

- Invalid updates are received such as updating a spaceship's fuel level past its capacity.
- If queried objects can't be found or don't exist.
- If a device is unavailable or becomes unavailable for some reason during attempts to reach and/or update objects.

Testing

Testing is done through a test driver class called TestDriver that contains a main method that accepts a command script file as a parameter. The script would have its own domain-specific language that corresponds to Flight Management service API method calls. The main method in TestDriver could exercise the Flight Management service API by reading in and parsing the script language and running the method calls. The TestDriver class should be set within the same classpath as the Flight Management service so that it can be run.

Risks

System performance may not be sufficient as it hasn't been tested. The module should be tested with the other components to ensure that it is running sufficiently well in terms of speed, hardware performance, and other important metrics.

The Manager might not be flexible enough based on the rules it follows. For instance, it would not know how to handle unexpected encounters with new undefined events. A possible update could include adding some machine learning or AI-driven layer to mediate this or adding new rules.

The Manager serves a critical function in the overall ISTS. It needs to be able to be online at all times. There should be more mechanisms in place to ensure this happens. For instance, it should always have access to a valid Auth Token for accessing the other services which may need extra function such as having a different expiry mechanism and that also isn't vulnerable to hacking. Further research on how to protect the Flight Manager's access vectors and other security risks is recommended so that measures can be implemented.

Abstractness and Instability Metrics

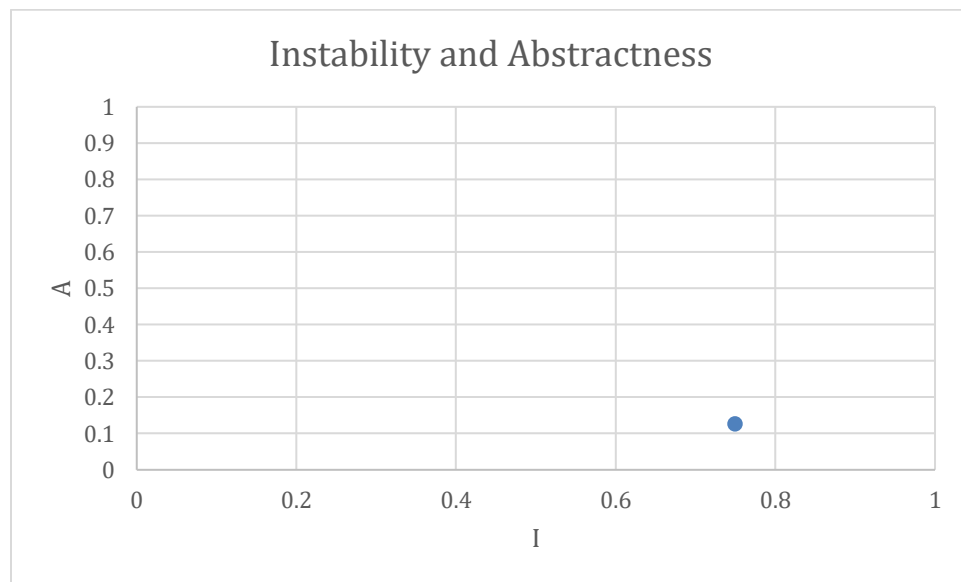
The afferent modules, C_a , for the Flight Management service are the GUI. The efferent modules, C_e , are the Customer service, Resource Management service, and Authentication service. This gives:

- $C_a = 1$ module
- $C_e = 3$ modules
- $\text{Instability} = I = C_e / (C_e + C_a) = 3 / (3 + 1) = 0.75$

The concrete and non-instantiable classes, N_c , for the Flight Manager include the interfaces and abstract classes (Action, and FlightManagementService), N_a , plus the concrete classes (Manager, LocationUpdateAction, StatusUpdateAction, EmergencyAction, ReachedDestinationAction, MessageAction, CompSystemAction, CommSystemAction, PassengerRegisterAction, MissionReportAction, DiscoveryAction, BoardPassengerAction, CustomerFeedbackAction, FuelLevelAction). This gives:

- $N_a = 2$
- $N_c = 16$
- Abstractness = $A = N_a / N_c = 2 / 16 = 0.125$

Below is the plot of Instability vs. Abstractness:



Caption: The plot of Instability vs. Abstractness metrics for the Flight Management System.

It the above chart, the design of the Flight Management service seems to lean more toward the main sequence than anything else but also towards being unstable and concrete. Its distance, D , from the main sequence:

$$D = |A + I - 1| = |0.125 + 0.75 - 1| = 0.125$$

GUI

Customer Service GUI

The Customer service GUI will be used by passengers and administrators of the ISTS. Once logged in, Administrators can click on navigational buttons on the screen in order to get to and from the Resource Management, and Flight Manager GUIs while non-administrative passengers

can only use the Customer service GUI. For administrators, the service will remember which GUI was being used at logout so they can pick up where they left off when they sign back in. When users first sign in to the service, the most up-to-date Customer service data for the user is automatically downloaded from the remote persistence storage location on the IPFS to local memory using the Customer service's "pullFromIpfsRepo" API method. Below is what the login/register form should approximately look like:

Login/Register Form

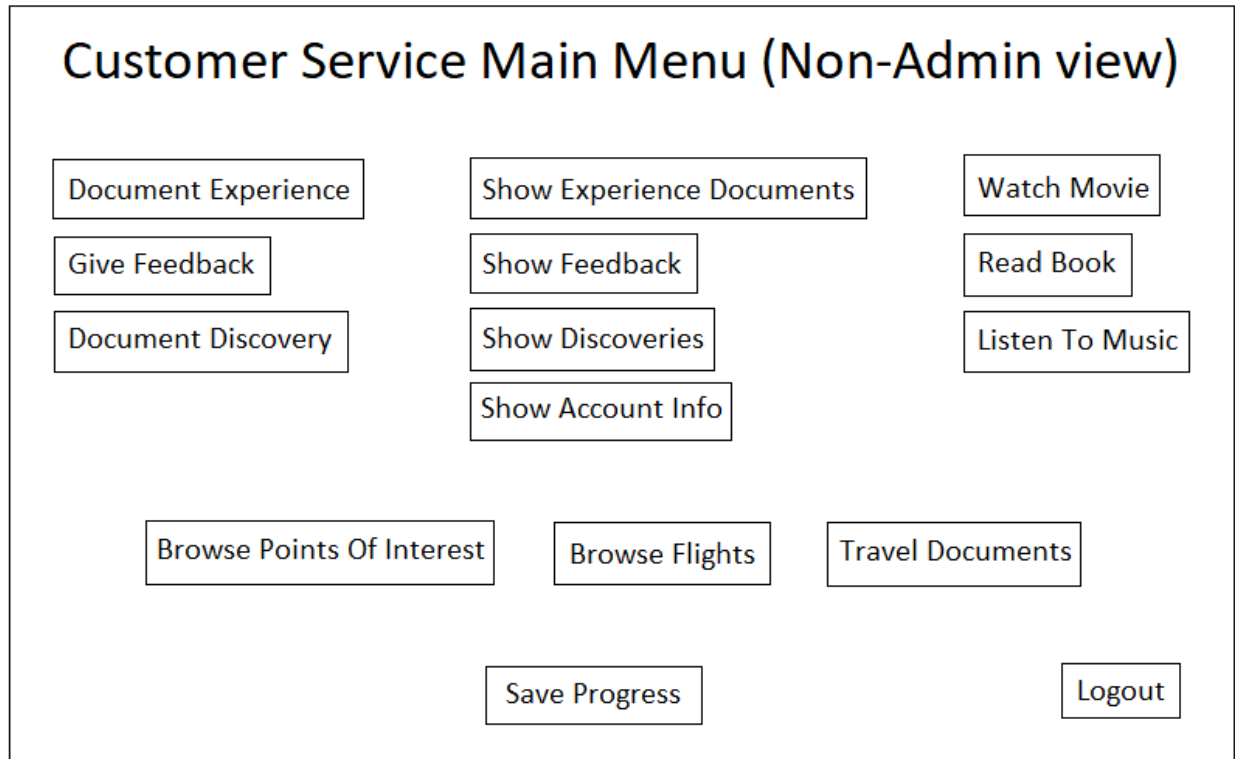
Username:

Password:

Click [here](#) to register.

Caption: The login/register form for the Customer service GUI. The "Click here to register." link at the bottom opens up a "Define <Customer Object>" form (defined later in this section) when clicked where passengers can fill out registration information to create an account to log in.

After users log in, they should be brought to the main menu. Below is what the main menu screen of the Customer service GUI should approximately look like:



Caption: The main menu screen of the Customer service GUI for non-administrative passengers. The Admin Customer service main menu GUI is shown at the end of this Customer service GUI section for comparison.

The top left column of buttons on the main menu that allow the user to create documents when clicked on bring up a form window that should look approximately like the following:

Define <Customer Object> Form

Id:

Attribute 2:

Attribute 3:

⋮
↓

Attribute n:

Caption: The generic “Define <Customer Object>” form of the Customer service GUI where n is the number of attributes. It allows users to create a document such as a discovery made on a trip by defining its attributes in a form and then hitting the save button.

After hitting the save button, the GUI will use the appropriate Customer service API method to create the new type of Customer object. For instance, if the form was for defining a new Discovery, then the “defineDiscovery” method would be called with the inputted attributes as parameters.

For Customer objects that contain media (e.g., Discovery), the “Define <Customer Object>” form should include media management functionality that should look approximately like the following:

Upload Media Function (in forms for objects with media)

Caption: The upload-media functionality that's included on forms for defining Customer objects that contain media, e.g., experience documentation can have video recordings. Clicking on a thumbnail opens the media in another window for an enlarged view and/or to be listened to or read. Uploading new media creates a new clickable thumbnail in the list for the media. Media can also be checked and deleted from the list.

The “Show <Customer Object>” buttons in the top middle column of the main menu shows the type of Customer objects and the information they contain when clicked on. For instance, “Show Experience Documents” should bring up a new form window with the passenger’s experience documents and information details that should look approximately like the following:

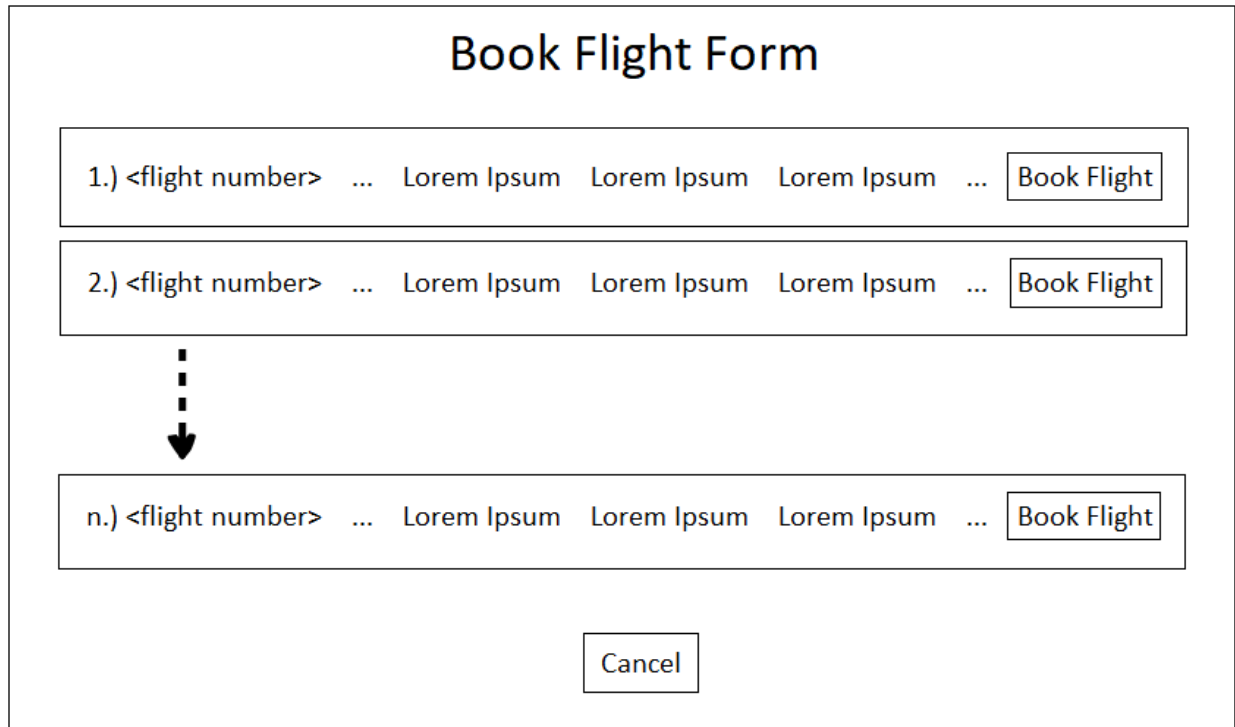
Show <Customer Object> Form

<p>1.)</p> <p style="margin-left: 40px;">Id: <customer object> id Attribute 2: <customer object> attribute 2 Attribute 3: <customer object> attribute 3 ... Attribute n: <customer object> attribute n</p> <div style="text-align: right; margin-top: 10px;"> <input type="button" value="Delete"/> <input type="button" value="Edit"/> </div>	<p>2.)</p> <p style="margin-left: 40px;">Id: <customer object> id Attribute 2: <customer object> attribute 2 Attribute 3: <customer object> attribute 3 ... Attribute n: <customer object> attribute n</p> <div style="text-align: right; margin-top: 10px;"> <input type="button" value="Delete"/> <input type="button" value="Edit"/> </div>	<p style="text-align: center;">...</p> <p>m.)</p> <p style="margin-left: 40px;">Id: <customer object> id Attribute 2: <customer object> attribute 2 Attribute 3: <customer object> attribute 3 ... Attribute n: <customer object> attribute n</p> <div style="text-align: right; margin-top: 10px;"> <input type="button" value="Delete"/> <input type="button" value="Edit"/> </div>
<input type="button" value="Cancel"/>		

Caption: The generic “Show <Customer Object>” form of the Customer service GUI where n is the number of attributes an object has and m is the number of that specific type of Customer object there is. It allows users to view, update, and delete Customer objects. When clicking on an edit button, the corresponding Customer service object’s “Define <Customer Object>” form should be brought up in a new window to allow for editing.

When objects are shown, the GUI utilizes the Customer service accessor API methods. For instance, to show all the user’s discoveries, it would need to call the “getDiscoveries” method.

The middle row of buttons (above the “Save Progress” button) in the main menu are for browsing points of interest and flights, booking flights, and accessing important travel documents and flight information after booking a flight. These can’t be edited by passengers; only read. However, passengers can upload their passport and visa information on the form that opens when they click on the “Travel Documents” button. The “Browse Flights” button brings up another window for booking flights that looks approximately like the following:



The diagram illustrates the 'Book Flight Form' layout. It features a title 'Book Flight Form' at the top center. Below the title, there are three horizontal rows of form elements. Each row contains a label (e.g., '1.) <flight number>', '2.) <flight number>', and 'n.) <flight number>'), followed by three 'Lorem Ipsum' placeholders, and a 'Book Flight' button. A dashed arrow points from the second row down to the 'n.' row, indicating a sequence or continuation. At the bottom center of the form is a 'Cancel' button.


Caption: The form for booking flights in the Customer service GUI where n is the number of flights available. It allows users to browse available flight information and book them by clicking the “Book Flight” button. When a user books a flight, the “bookFlight” Customer service API method is called with the listed parameters.

The top right column of buttons in the main menu allow passengers to access in-flight entertainment. These can’t be edited by passengers; only read. Clicking on them should bring up a form that looks approximately like the following:

Access <Media> Form

1.) <media name> ... Lorem Ipsum Lorem Ipsum Lorem Ipsum ... Access <Media>

2.) <media name> ... Lorem Ipsum Lorem Ipsum Lorem Ipsum ... Access <Media>



n.) <media name> ... Lorem Ipsum Lorem Ipsum Lorem Ipsum ... Access <Media>

Cancel

Caption: The generic form for allowing users to pick media out from a list for entertainment purposes during the flight where n is the number of list items. Users can click on the “Access <Media>” button to stream the media. Media offered includes movies, music, and books. The appropriate accessor/getter API method is called per the type of media being accessed.

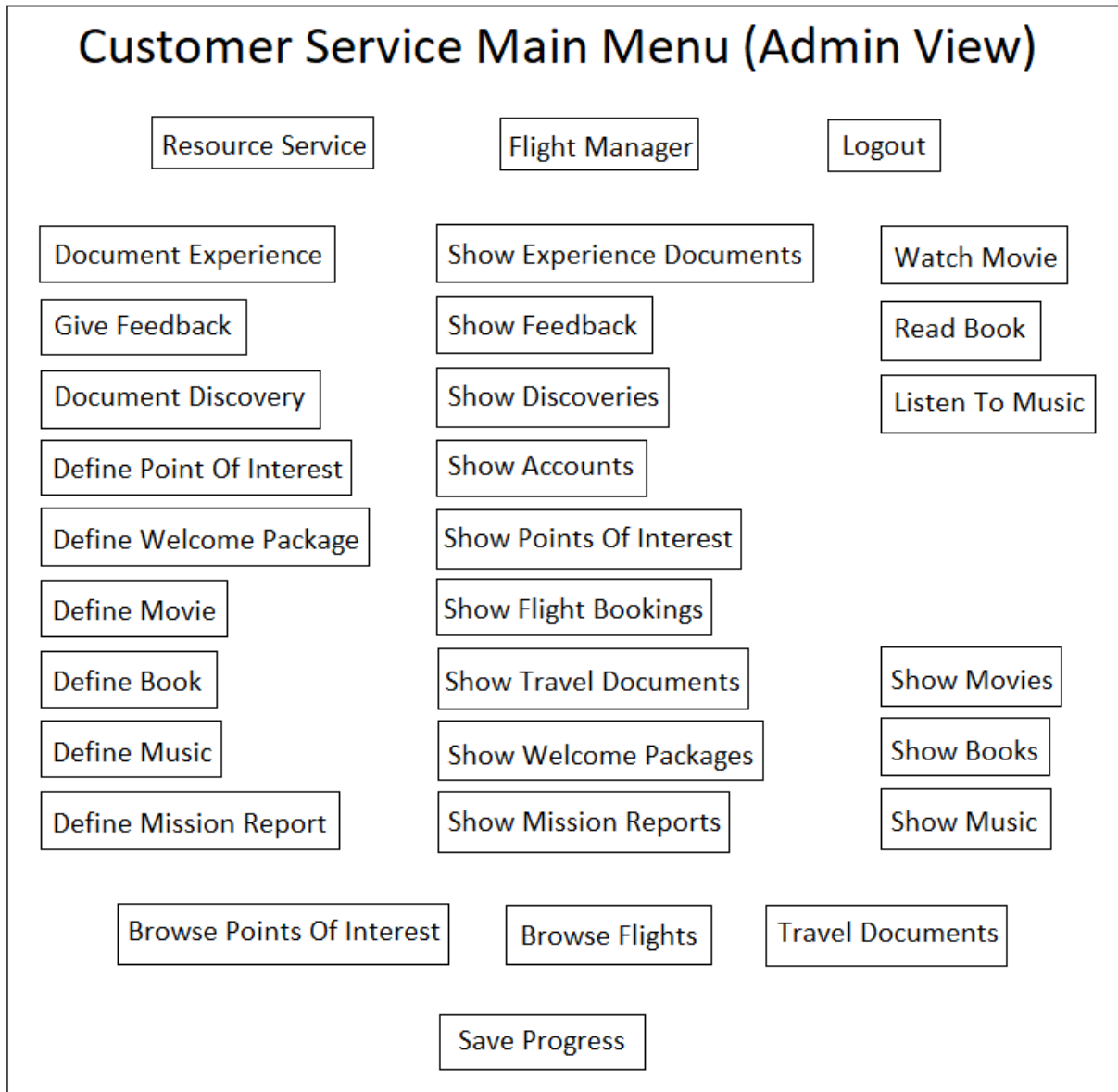
The bottom “Save Progress” button on the main menu uploads any local data updates/changes made in local memory to the remote IPFS for persistence, so that it can be retrieved again at a later time. This calls the “pullFromIpfsRepo” and “pushToIpfsRepo” Customer service API methods in succession.

Admin Customer Service Main Menu GUI

The Admin view of the Customer service main menu GUI is the same as Non-Admin passenger’s except for the following:

- Admins not only get back a view of their own data but every passenger’s when showing experience documents, feedback, discoveries, and accounts.
- Admins can navigate to other services (Resource and Flight Manager).
- Admins can define more types of Customer objects and edit them than regular passengers can, e.g, points of interest, travel documents, in-flight entertainment, welcome packages, and mission reports.
- Admins can view (and edit) all flight booking records.

The following is what the Admin’s main menu should approximately look like:

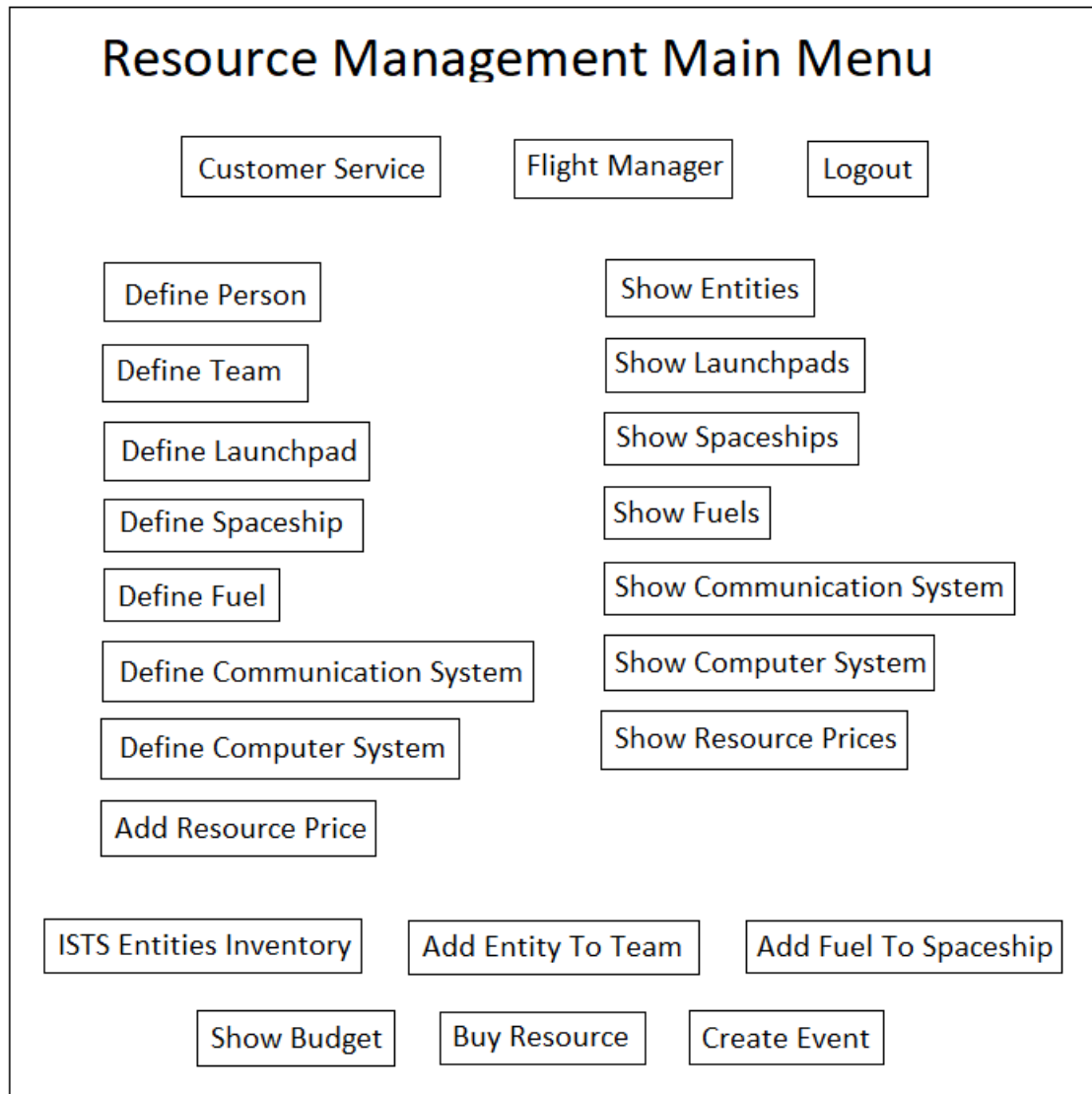


Caption: The main menu screen of the Customer service GUI for Admins. Admins are able to have their own personal user objects and also act administratively over all objects. For instance, the “Travel Documents” button accesses the Admin’s personal user travel documents while the “Show Travel Documents” button would return every ISTS passenger’s travel documents since the Admin is allowed to read, update, and delete any of them. For another example, Admins can watch movies by clicking the “Watch Movie” button and also update and delete movies by clicking the “Show Movies” button or define a new movie by clicking the “Define Movie” button.

Resource Management Service GUI

The Resource Management GUI will only be used by administrative users of the ISTS (refer to the beginning of the Customer service’s GUI section for an explanation of how logging in to the service and navigation works). Below is what the main menu screen of the GUI should

approximately look like:



Caption: The main menu screen of the Resource Management service GUI.

The middle left column of buttons on the main menu that allow the user to define the state of each of the resources when clicked on bring up a form window that should look approximately like the following:

The diagram shows a form titled "Define <Resource> Form". It contains four input fields, each with a label and a placeholder text: "Id: enter <resource> id here", "Attribute 2: enter <resource> attribute 2 here", "Attribute 3: enter <resource> attribute 3 here", and "Attribute n: enter <resource> attribute n here". A dashed arrow points from the "Attribute 3" field down to the "Attribute n" field. At the bottom of the form are two buttons: "Save" and "Cancel".

Caption: The generic “Define <Resource>” form of the Resource Management GUI where n is the number of attributes. It allows users to create a resource by defining its attributes in the form and then clicking the save button which would call the resource’s “define<resource>” API method with the inputted attributes as parameters.

The middle right column of buttons on the main menu that show the type of resource objects and the information they contain when clicked on bring up a form window that should look approximately like the following:

The diagram shows a form titled "Show <Resource> Form". It contains three panels, each representing a resource object. The first panel is labeled "1.)" and contains the text: "Id: <resource> id", "Attribute 2: <resource> attribute 2", "Attribute 3: <resource> attribute 3", "...", and "Attribute n: <resource> attribute n". Below this text are two buttons: "Delete" and "Edit". The second panel is labeled "2.)" and contains the same text. The third panel is labeled "m.)" and contains the same text. There are three dots between the second and third panels. At the bottom of the form is a "Cancel" button.

Caption: The generic “Show <Resource>” form of the Resource Management GUI where n is the number of attributes the resource object has and m is the number of that specific type of resource there is. It allows users to view, update, and delete resources. When clicking on an edit button, the corresponding resource’s “Define <Resource>” form should be brought up in a new window to allow for updating.

The bottom two rows of buttons in the main menu involve functions that perform or provide support for some type of behavior on or manipulation of resource data. For instance, the “ISTS Entities Inventory” button brings up a window that shows an organized form view of the people and teams that make up the ISTS organization. This could be used to organize groups of people, e.g., flight crews and passengers, in order to support managing flights. More interactive functionality could also be included.

The top row of buttons on the main menu are for navigational purposes such as directing users to other ISTS services or logging out.

The “Create Event” button on the main menu, brings up a form that allows the user to create/simulate an event by typing in an event with the appropriate event command-line syntax. The event and resulting actions are logged.

Flight Management Service GUI

The Flight Management GUI will be used by administrators of the ISTS (refer to the beginning of the Customer service’s GUI section for an explanation of how logging in to the service and navigation works). Below is what the main menu screen of the GUI should approximately look like:



Caption: The main menu of the Flight Management service GUI.

The main menu of the Flight Management service is relatively simple. It’s a very behavioral service and an administrator would need to navigate to the other services’ GUIs in order to monitor and update objects there. The top row of buttons navigates to the service GUI that’s clicked on.

To manage resources, an administrator would navigate to the Resource service GUI and use its interface (see the Resource Management service main menu GUI defined previously). There, they could click the “Show Spaceships” button which would bring up a list of all the ISTS spaceships and their information (see the “Show <Resource>” form in the previous Resource service GUI section for how this would appear). An administrator could then read, update, and

delete any information of any spaceship. This includes the messages that a spaceship has, fuel level, and the number of available spacecraft. The Resource service GUI also allows for access to the communication system (by clicking the “Show Communication System” button), the operating budget (by clicking the “Show Budget” button), and more.

To manage flight status, an administrator would navigate to the Customer service GUI (see the Customer service main menu GUI defined previously). There they could click on the “Show Flights” button which would bring up a list of all the ISTS flights and their information (see the “Show <Customer Object>” form in the previous Customer service GUI section for how this would appear). An administrator could then read, update, and delete any information of any flight including flight status.

The “Log” button on the Flight Management main menu should print to stdout a log of the events and actions that have happened in the automated control system when clicked on by calling the “getLog” method on the Flight Management’s API.

One of the most important functions of the Flight Management main menu GUI is to support flight creation. When the “Define Flight” button on the main menu is clicked, a form window should be brought up that should look approximately like the following:

Define Flight Form

Id:

Flight Number:

Spaceship:

list of spaceships

Time:

Location:

list of launchpads

Destination:

list of destinations

Duration:

Num Stops:

Capacity:

Crew :

list of crews

Ticket Price:

Passenger Count:

Ok

Cancel

Caption: The “Define Flight” form of the Flight Management GUI. It allows administrators to create a new flight by defining its attributes in the form and then clicking the Ok button which would call the Flight Management API’s “defineFlight” method with the inputted attributes as parameters. It includes a drop-down list function for some attributes for selection convenience.

For the above “Define Flight” form to be created, the GUI would call the Flight Management service’s “getFlights”, “getSpaceships”, “getEntitiesVisitor”, “getLaunchpads”, and “getPointsOfInterest” API methods in order to get all the external objects that are available and necessary, or helpful for creating a flight. For instance, the GUI would need to retrieve all available spaceships in order to populate the form’s list of spaceships for selection.