Interplanetary Space Transport System (ISTS) Architectural Overview

Date: 12/18/2019

Author: Gerald Arocena

Reviewer(s): Krithika Sundararajan

Whole document structure overview/explanation here.

# 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. 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 (GUI) 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 Assigment 4 must also be used to check access permissions for the GUI and service APIs including 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 documention 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 applied and highlighted where appropriate.

Persistence

The ISTS will use the InterPlanetary File System for sharing documents and long-term storage. Is 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.

# 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 even though it’s not convention to include, the inclusion of accessors and mutators in this document was deliberate in order to highlight the importance of their access permissions and/or the importance of their function to the overall IST system. For instance, I chose to put some important methods on the domain objects themselves instead of, say, on a service’s API so it’s important to be able to retrieve objects like these. In addition, my implementation of the Authentication Service used a class called AuthTokenTuple (which includes an AuthToken) and is included in this document. Please refer to my Authentication Service implementation for further clarification on its use.

ISTS Resource Management Service Design Document

Document structure overview here.

# Introduction

This design document 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 a 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 emitted by spaceships and the communication system.

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

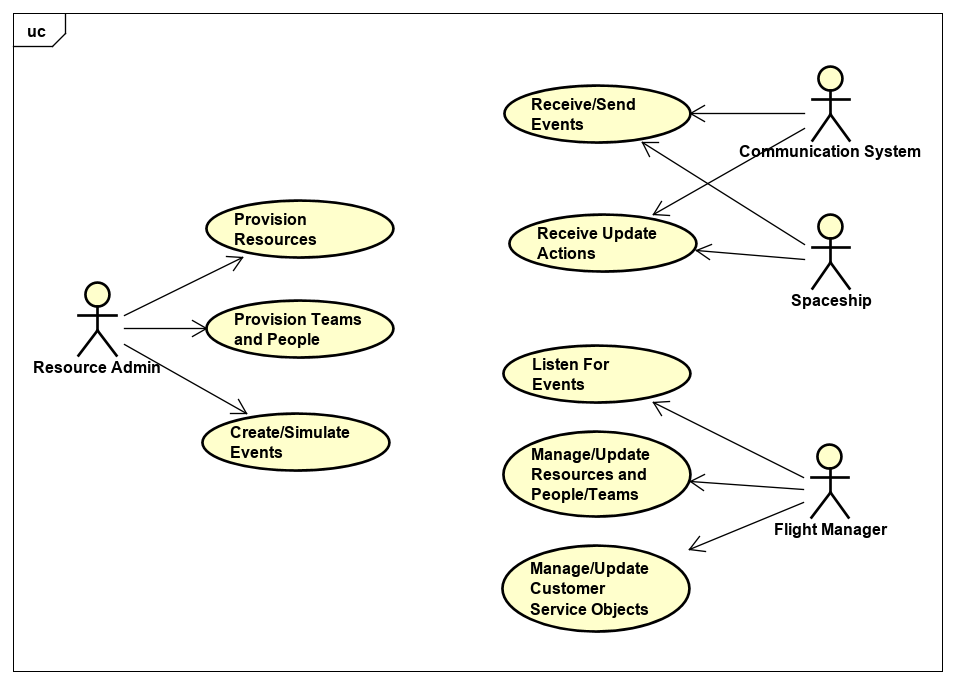
# Requirements

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 account since it has a budget resource that must be used 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 System might depend on the Resource Service to find out a spaceship’s capacity for flight creation. It will also be listening for interesting events in the Resource Management Service in order to manage and update its resources. Other resources include fuel, communication system, and computer system.

The Resource Service must also be able to create spaceship and communication system events. Spaceships must be able to receive update actions in response to events such as when a flight reaches its destination. For this module’s design document, events can be thought of as opaque strings. Finally, all API methods must accept an AuthToken parameter to support access control.

# Use Cases

The diagram below illustrates the uses 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. It listens for interesting events emitted by the Resource Management Service and responds with an appropriate action including requests to the Resoure 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 an intermediate agent in the communication system of the ISTS. It is utilized in creating/simulating events.

**Spaceship**

The spaceships are able to communicate update events about their status or the status of the system to the Resource Management Service.

**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. These should take the form of formatted and translatable messages.

**Receive Update Actions**

Update actions should be able to be sent through the communication system (e.g., to the spaceship and Resource Management Service to be updated by external automatic control systems, e.g., the Flight Management Service in response to events).

**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.

**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.

# Class Diagram

The following Resource Management Service class diagram defines the classes defined in this design that comprise the package “com.cscie97.ists.resource” as well as classes from other packages that have an important relationship to the Resource Management Service.

Class diagram here.

Caption: Class diagram for the Resource Management Service.

# 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. |
| defineCommunicationSystem | (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 is the amount of fuel. |
| 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. |
| 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 document).

***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 (discussed earlier in this dictionary) implementation Singleton class that also implements the Visitable interface discussed above.

***Properties***

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

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| entities | map<entityId, Entity> | The entities (extended by Person and Team) that make up the ISTS organization hierarchy. |
| launchpads | map<launchpadId, Launchpad> | The launchpads that spaceships use. |
| spaceships | map<spaceshipId, Spaceship> | The spaceships used by the ISTS. |
| communicationSystem | CommunicationSystem | The communication system used by the ISTS. |
| computerSystem | ComputerSystem | The computer system used by the ISTS |
| 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 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. 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. |
| addPassengerId | (passengerId : string, authTokenTuple : AuthTokenTuple) : void | Adds the given passenger id to the list of id’s 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 trajectory of the spaceship. |
| coordinates | string | The coordinates of the spaceship. |
| crewCapacity | int | The crew capacity of the spaceship. |
| passengerIds | string[] | The Passenger id’s of the passengers on the spaceship. |
| currentFlightId | string | The id of the flight the spaceship is currently associated with. |

***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 resourceImpl’s 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), 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, authTokeTuple : 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, authTokeTuple : 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 up 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 for interfacing with spaceships. |

***Properties***

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

## ComputerSystem

This class represents an abstraction for large computer facilities including 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. |

## 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. |

# 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, the Spaceships hold a list of passenger id’s 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 id’s in a namespace such as with Entity id’s.
* 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 reources. It is not required but perhaps the implementation can be updated to make use of the IPFS like the Customer Service module does for long-term storage.

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.

Also, 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 such a frontier endeavor.

ISTS Customer Service Design Document

Document structure overview here.

# Introduction

This design document 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 utilizes the InterPlanetary File System (IPFS) for sharing files.

Please refer to the component diagram in the Architectural Overview section 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 (e.g. Credential class from Authentication Service). 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 (e.g., travel tickets, welcoming information packets, passports, and visas), in-flight entertainment (e.g., movies, music, and books), and discoveries and mission progress (e.g., 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. 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 accesible to passengers to stream during the flight. And 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 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 interaction 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 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 AuthToken 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; they can only access and update 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 an account with the IPFS and login and logout 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. 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 UI. The should also be able to be recognized by the ISTS from a voiceprint and faceprint Credential. The Authentication Service could 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 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 action.

**Provision Flights**

The Flight Manager is reponsible for provisioning the flights of the ISTS.

**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 them.

**Book Flights**

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

**Access Travel Documents**

Passengers and 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 they made on the flight and mission reports. For example, a researcher passenger could record a newly discovered life form on a waypoint along 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 Manage needs to save discoveries and mission reports to the IPFS in response to a Reached Destination flight event.

**Create Events**

The Customer Service should 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 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 document.

# Class Diagram

The following Customer Service class diagram defines the classes defined in this design that comprise the package “com.cscie97.ists.customer” as well as classes from other packages that have an important relationship to the Customer Service.

CLASS DIAGRAM GOES HERE

Caption: Class diagram for the Resource Management Service.

# 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** |
| pullFromIpfsRepo | (authTokenTuple : AuthTokenTuple) : void | Fetches and downloads all 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 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 with the given parameters and adds it to the map of passengers. Account is their Ledger Service account. |
| getPassengers | (authTokenTuple : AuthTokenTuple) : map<passengerId, Passenger> | Returns the map of registered passengers. |
| addPassengerCredential | (passengerId : string, type : string, value : string, authTokenTuple : AuthTokenTuple) : void | Creates a new Credential for the passenger of the id given with the given parameters and adds it to the passenger’s map of credentials. |
| 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 document 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, visaId : string, welcomePackageId : 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 to the passenger’s boarding pass image 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<welcomePackageId, 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, messge : 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 (discussed above) implementation Singleton class.

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| hashCalculator | (originalString : string) : string | Utility method that hashes a string. Used for credentials. |
| 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 the remote IPFS Customer Service document database. |

***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. |
| authenticator | StoreAuthenticationService | A reference to the Authentication Service Singleton. |
| passengers | map<passengerId, Passenger> | The map of passengers. |
| credentialUserIndexes | map<id, Passenger> | The mapping of concatenated credential id’s and values to their corresponding passenger account (for credential checking). |
| 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 a registered 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. |

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| credentials | map<credentialId, Credential> | The credentials of the passenger, i.e., username/password, voiceprint, and faceprint. |

## 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 take 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<noteId, 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<imageId, Image> | The map of images. |
| movies | map<movieId, 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 can be uploaded 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 can be uploaded 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. |
| boardPassIpnsKeyName | string | The IPNS key name that links to a picture 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** |
| 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. |

# Implementation Details

The Customer Service handle’s everything customer service related in the IST system. The key actor in this component is the passenger in how they interact with the IST system. It provides a flight booking system that’s similar in function to current 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. One way to do this is to add the passenger as a Person in the Resource Service when they register with the Customer Service. The Customer Service also needs a reference to the Resource Service because it can also create events and so it needs to utilize the communication system to, for instance, notify observers (i.e., the Flight Manager) of interesting mission reports, or discoveries.

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 necssarily 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 the bootstrap process (perhaps in the constructor of the Customer Service) or some other method soon after 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 see anyone’s data). To have privacy control, the Customer Service should use its methods’ AuthToken parameter to only return data that a user that’s calling a method has rights to. And to facilitate this process, the Customer Service could associate an object to a particular passenger by systematically giving the domain objects id’s that include the passenger id in it. For instance, a video recording given the id “passenger 1:videoRecording 1” would indicate that that particular video recording belongs to passenger “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 and passenger registrations could be automatically uploaded to the IPFS after they’re created as they are of relative importance. It would also be a good idea to download the latest state of the remote IPFS data to a user’s local memory when they sign in to the Customer Service portal after a period of inactivity.

The Customer Service also has important dependencies on other components in the ISTS. For example, consider the following sequence diagram for booking flights:

Sequence diagram goes here.

Caption:

# 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 id’s.

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, passenger accounts, flight bookings, and documents. It is not mentioned in the requirements but perhaps the implementation can 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 overheard 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.

ISTS Flight Management Service Design Document

Document structure overview here.

# Introduction

This design document explains how to implement the Flight Management Service module of the IST system. The Flight Manager

Overview

…

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

# Requirements

*This section provides a summary of the requirements for the <Component Name>.*

*Provide your understanding of the requirements, both functional and nonfunctional. Reference the provided Requirements and System Architecture documents. Do not cut and paste from the requirements document.*

*Product Manager and others can read this to understand what requirements your design will support. There is already a requirements doc, so keep this brief and to the point, highlighting the important requirements that the design is addressing. Structure in a way to provide a requirements checklist for your design.*

# Use Cases

*Enumerate the use cases supported by the design,*

*This design supports the following use cases:*

*Include a Use Case Diagram.*

*Include descriptions of each of the actors and use cases.*

# Implementation

*This section of the document will describe the implementation details for ...*

*The implementation section should cover the following topics:*

* *What are the classes, and their properties, associations and methods?*
* *What are the important interfaces and how they will be implemented?*
* *How are the requirements addressed?*

# Class Diagram

…as well as classes from other packages that have an important relationship to the Flight Management Service.

*CLASS DIAGRAM GOES HERE*

Caption: For example, the Customer Service class isn’t fully filled in because not all of it is core to understanding how the automated management process works.

# Class Dictionary

This section specifies the class dictionary for the 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)

*Class 1 description*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| defineFlight | (id : string, number : string, spaceshipId : string, time : string, location : string, destination : string, duration : string, numStops : int, capacity : int, crewId : string, ticketPrice : int, passengerCount : int, authTokenTuple : AuthTokenTuple) : Flight |  |

## Observer (Interface)

*Class 1 description*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| update | (event UpdateEvent): void |  |

## Manager

Implements FlightManagementService and Observer interfaces.

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| handleEvent | (event : UpdateEvent) : void |  |

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| resourceImpl | ResourceManagementService; Subject |  |
| customerImpl | CustomerService | Reference to the Customer Service which the Manager performs update actions to and provisions flights for. |
| authenticator | StoreAuthenticationService |  |
| authToken | AuthToken | The Manager needs an AuthToken with the proper access permissions to interface with the other modules 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”. |

## Subject (Interface)

*Class 1 description*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| registerObserver | (observer : Observer) : void |  |
| deregisterObserver | (observer : Observer) : void |  |
| notifyObservers | (sourceDevice : Spaceship, eventToSend : string[]) : void |  |

## ResourceImpl

Implements Subject

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| createEvent | (spaceshipId : string, simulatedEvent : string, authTokenTuple : AuthTokenTuple) : void |  |

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
| observers | Observer[] |  |
| spaceships | map<spaceshipId, Spaceship> |  |
| communicationSystem | CommunicationSystem |  |
| authenticator | StoreAuthenticationService | Used to check access permissions on the ResourceImpl object’s methods. |

## CommunicationSystem

*Class 1 description*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| createEvent | (sourceDevice : Spaceship, simulatedEvent : string) : string[] |  |

## Spaceship

*Class 1 description*

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| event | (perceivedEvent : string) : string[] |  |

## 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 through the communication network (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 sent to a spaceship, and the Resource Management and Customer services. It is an abstract class that each event extends with a corresponding Action subclass. When a new Action is made it is executed right away.

***Methods***

|  |  |  |
| --- | --- | --- |
| **Method Name** | **Signature** | **Description** |
| execute | () : void | Defines the executable actions that are 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. |
| resourceImpl | ResourceManagementService | A reference to the Resource Management Service that the self-sufficient Action classes depend on in their execute methods. |
| customerImpl | CustomerService |  |
| manager | FlightManagementService |  |
| authToken | AuthToken | To perform their functions the Action classes need an authToken with the appropriate access permissions. |

## EmergencyAction

Extends Action.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| emergencyType | string |  |

## StatusUpdateAction

Extends Action.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| status | string |  |

## ReachedDestinationAction

Extends Action.

## LocationUpdateAction

Extends Action.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| speed | int |  |
| trajectory | string |  |
| coordinates | string |  |

## FuelLevelAction

Extends Action.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| fuelLevel | int |  |

## MissionReportAction

Extends Action.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| missionReportInfo | string |  |

## DiscoveryAction

Extends Action.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| discoveryInfo | string |  |

## CustomerFeedbackAction

Extends Action.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| feedbackInfo | string |  |

## BoardPassengerAction

Extends Action.

***Properties***

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Type** | **Description** |
| passengerId | string |  |

## Gui

Class 1 description

***Associations***

|  |  |  |
| --- | --- | --- |
| **Association Name** | **Type** | **Description** |
|  |  |  |

# Implementation Details

* The brains of the system
* In charge of flights including in emergency situations

*Explain details of the implementation.*

*How do the various parts fit together or interact?*

*How does the design address the requirements? Justify your design decisions and how they address the requirements.*

*Some implementation details may be addressed in the class dictionary, but for things that are not, describe them here.*

*Remember to reference the requirements from the body of the design document to show how your design is addressing the requirements.*

# Exception Handling

*Provide details on your exception handling. What types of exceptions are expected and how are they handled by the design? Describe your exception classes and their properties.*

# Testing

*Provide a testing strategy for testing the component.*

* *Functional*
* *Performance*
* *Regression*
* *Exception Handling*

# Risks

* Keeping the Customer Service logged in perpetually
* Validating parameters and data before submitting to IPFS