Chalkboards Open API

NOTE: Truncated version, full version available on sign-up

Staff

1/1/2018

Developer’s guide for Chalkboards API.

CONTENTS

OVERVIEW

Chalkboards Open API is designed to support building applications which require many common functionalities typically required by small businesses.

The API is modular in design and therefore extensible.

This document is intended for external developers using the Chalkboards API.

We define several terms that will be used throughout the document.

TERMINOLOGY

SERVICEACCOMMODATOR

Businesses can usually be identified by their ownership entity and their physical location. We call the ownership entity as the ServiceAccommodator. For example, Joe's restaurants have many locations, but may share a corporate entity that form the ownership (or owner of the Brand). In the case, the Joe's corporation would be a ServiceAccommodator.

SERVICELOCATION

The physical location or branch of a business is called the ServiceLocation in the system. The same ServiceLocation could be shared by multiple businesses, although this is unlikely. The ServiceLocation is also the registered or official location of a business branch. The current location is saved in a separate table.

SERVICEACCOMMODATORSERVICELOCATION (SASL)

The combination of unique ServiceLocation and ServiceAccommodator identifies business location in the system. The system continues to maintain all physical attributes against the ServiceLocation, while all ServiceProviders (employees) and Services continue to be maintained against a ServiceAccommodator. This design allows, if applicable, the sharing of employees (and their services) across multiple locations, with respect to reservations and appointments. Retail catalogs are also maintained in a similar manner allowing items to belong to the organizational catalog or branch catalog and retrieved in a branch specific manner.

RETAIL

Retail services are defined as all services related to retail sales, either instore via a point-of-sale or online via e-commerce websites or apps. Retails catalogs are multilevel data structures that are responsible for holding all items for sale in a manner that accommodates versioning by size, color etc., as price changes maintaining pricing history. Orders, Invoices etc., are defined in the usual manner, as commonly defined in standard accounting practices. However, as a multi-channel API, order source (mobile, in-store, etc.) is also maintained as part of the order meta-data.

RESERVATION

A reservation, for the purpose of this API, is distinguished as a Booking or planned usage of a ServiceLocation facility. ServiceLocation have Floors, Tiers, Seats (such as tables in a restaurant, or 'bay' in a auto workshop). A reservation always refers to the planned usage of such a physical resource, although a service or serviceprovider may be involved, but it is assumed their time is not relevant. A restaurant booking is therefore a reservation, since it is against a table.

APPOINTMENT

An appointment, on the other hand, is a planned consumption of a service of a servcieprovider, and it is made against the time of a service provider. Thus, an appointment with a dentist may ignore the schedule of a visiting room, although a room will still have to be accounted for in the system.

Distinguishing between Reservations and Appointments allows us to have People centric or Physical Resource centric policies.

USER

We recognize a User as a person who is unique. A user can have an owner relationship or member relationship or service provider relationship with an SASL.

COMMUNITY

The term Community is used to describe the relationship of an SASL with users/members, and with the system itself as well as the state of an SASL with respect to the system. For example, an SASL can be cloned and can exist in multiple copies, but only one copy can be active at any time. This allows accommodating different versions of a business entity in different states. For example, we can distinguish between paying customers who have richer SASLs, which replace the default SASL that the system provided as a place-holder. An SASL (business) could be suspended, during which time the placeholder could again be active. A user maybe a member at more than one SASL (member relationship), while a person can own more than one SASL, and a person can be an owner in one SASL, and a member in another SASL and a service Provider in yet another SASL.

Sets of SASLs, which represent the same business, are identified by a unique URL-Key (a UUID that can be used a part of a URL). All external references to a business are made via such a unique URL key. The system will dereference the URL key at run-time based on which SASL is active at the moment. This URL-Key based system allows the system manage the SASL copy which publicly visible at any moment. Direct access to an SASL via the ServiceAccommodator and ServiceLocation IDs is not possible from outside.

LIVEUPDATE

SASLs can update their Service and Communication status at any time or their current location. Such updates are considered transient and usually have a short lifespan before they are considered expired. LiveUpdate collects such information into a group API entry points.

The following terms are listed without further explanation as they are more self-explanatory.

PROMOTION

Promotions are defined as item price changes or generic discounts applicable to entire catalogs or partial catalogs, and can be in the form of both absolute dollar (or currency) amount or as a percent of the price.
Promotions usually consist of a media (picture), title, description etc., as well as expiration, activation dates, and other meta-data. A promotion can be associated with a retail item, a retail catalog or un-related to any item (non-purchasable promotions).
Promotions have their own associated OG tags and OG compatible media (different resolution) to be social media sharing compatible.

MEDIA

Media is defined as generic media associated with any SASL that can be used for describing the business or introducing a business. At the moment, media comes in two versions, small and large, as well as a base64 encoded thumbnail, so that thumbnails can be embedded in API responses obviating the need for separate API calls to retrieve them.
Video media is also supported via links to CDNs. Our API does not support video hosting yet. A developer must get the retrieval API calls from the CDN where the video is hosted, either as a complete iFrame or as a video only link that can be embedded in

COMMUNICATION

Communication is defined as all forms of human involved communications (as opposed to computer-to-computer communications). The communication modules/APIs encompass email/SMS or text, native app notifications, as well as our own chat APIs.

CONTESTS

We support a variety of interactive polls, surveys, etc., which we collectively call Contests. Under contests we have polls with ‘prizes’, that can be awarded by a business owner via randomized draws. These primarily intended for promotions/interactive Ads, but can also be used for simple (single level) surveys.
The polling APIs also include timed ‘ending’s where users can bet against the system (non-gambling) for entertainment purposes. Selfie-contests allow users to upload pictures. Check-in contests require users to be within a geo-fence in order to enter.

API COMMON DETAILS

API DESIGN

The Chalkboards API is designed with REST (Representational state transfer) as an inspiration but it is not Restfull. In other words, we do not follow the REST recommendation of making everything a resource and design our API to be state transitions of these resources too strictly. When we can identify a resource whose state transition has business value, we do implement the various APIs necessary for a proper REST interface. Otherwise we switch back to Remote Procedure Call paradigm, where an API executes procedures (methods) on the server. We do use the HTTP verbs (GET,PUT,POST,DELETE) in the API. All retrieval must use GET, modification must use PUT, creations must use POST, and so on. The HTTP verb is part of the API signature matching mechanism, and failure to use the proper verb results in the server not recognizing the API being sought.

DATA INTERCHANGE FORMAT

Our data-interchange format is JSON (Javascript Object Notation), however, we use Multipart MIME types to include binary data where necessary, resulting in REST/JSON Multipart protocol.

ERROR PROPAGATION

Finally, for error propagation, we use our own simplified mechanism. We use Java exceptions as our main exception mechanism, with only Exceptions defined and used throughout the system. UnableToComplyException, which indicate an exception resulting from inappropriate or insufficient parameters etc., while, a PanicException indicates inconsistency or unexpected results in a server. These Java Exceptions are transmitted to the client using a combination of HTTP status code and predefined JSON objects (400, 500). We choose such a highly simplified mechanism so that we can traverse language barriers (all the way from database to Apps) in a consistent manner. A simplified system obviously results in some loss of fidelity, but the simplicity results in a practical and easy to use system.

CORS RESPONDER

Chalkboards supports a Cross Origin pre-flight authorization server. This allows modern browsers to allow complete HTTP access to our APIs from documents loaded from other hosts (hence API access would be considered cross-origin). For IOS/Android Apps, there is no Cross Origin restriction, and hence this is only relevant for WebApps or Web sites utilizing our API.

AUTHENTICATION

Authentication is defined as all functions related to verifying user identity. Chalkboards uses a modified OAuth2 as well as standard OAuth2 authentication mechanisms. We recommend the modified OAuth2 as it is designed to provide forensic data in the event of any attempted hacking, which, in turn serves as a dis-incentive to would-be hackers.

Chalkboards API do not require any PII (Personally Identifiable Information) for user accounts, although such information may be required for financial transactions. In such cases we usually defer to the credit-card processing

vendor to manage such information. Our credit processing partners are Worldpay (Vantiv) and PayPal. Other vendors may be added on request.
Internally Chalkboards APIs use a time-sensitive generated UUID assigned to all users and dereferenced via the authentication module when necessary. This allows users to change any authentication details, like 'username' or 'email' etc.. The UUID itself is multipart, with an embedded timestamp. This allows the UUID itself to be changed from time to time on a protocol negotiated between the client and the server. This prevent identity theft and other forms of snooping attacks etc., to a great extent.

We do not track users, unless that is part of the business function, and the user has provided explicit permission.

LOCATION

One of the services provided by Chalkboards is the location management of an entity (SASL).

1. Proximity Searching Given a set of SASLs whose locations are known (either dynamic or static), the API can search for such SASLs by proximity to a given location. This is useful for 'looking up' businesses near a user, for example. A third party 'map' will be required if such locations need to be displayed on a map. There are many map APIs available, with various forms of restrictions and the client must decide which one to use. Usually, the map itself is downloaded directly by the client, with the search results from our service overlaid on the map.
2. Geo Fencing: This is the ability create arbitrary regions joined by geographic location points, and determining whether a given location (such as the user) is inside or outside such a closed path. Modern telephones usually provide geo-fencing built-in as part of modern operating systems. However, sometimes, for reasons of privacy or security, we may need to use a server based geo-fencing mechanism.
3. Geo Coding: This refers to the determination of a street address given a location, and vice-versa. The system does not provide such service directly as it does not have any map data. However, it can access publicly available API and provide a single entry point for complex functions, if that is the requirement.

Location services are provided by using the Military Grid Reference System as the indexing framework. The accuracy can thus be varied between 1 meter to 100 k meters. Higher accuracy will result in slightly higher latency in API calls, -depending on any additional calculations necessary.

RELATIONAL MODEL

At the core of the system is the Relational Data Model (Chalkboards Schema) that breaks down the various aspects of typical e-commerce and mobile applications into a relational model. This is a very generalized model and as such consists of many details which may be irrelevant or not-applicable in the instance of a specific business scenario. In such a case the model is designed to degenerate to a simpler model and thus continue to accommodate that particular industry. How this is achieved will be described in more detail in the section of the Chalkboards Relation Model.

Layered on top of this relational model is the ORM layer (Object Relational Mapping) which utilizes an industry standard JPA implementation (e.g. Hibernate).

The Java Logic layer uses this ORM layer and forms the core logic of the system. Here specific business rules can be implemented which in turn will use the underlying relational model to determine the logical conditionals as given for any specific business. Combining the logic and the business specific conditionals provides a powerful logic layer that can be very flexible and very business specific. There can be multiple logic layers, each specific to a particular industry or business.

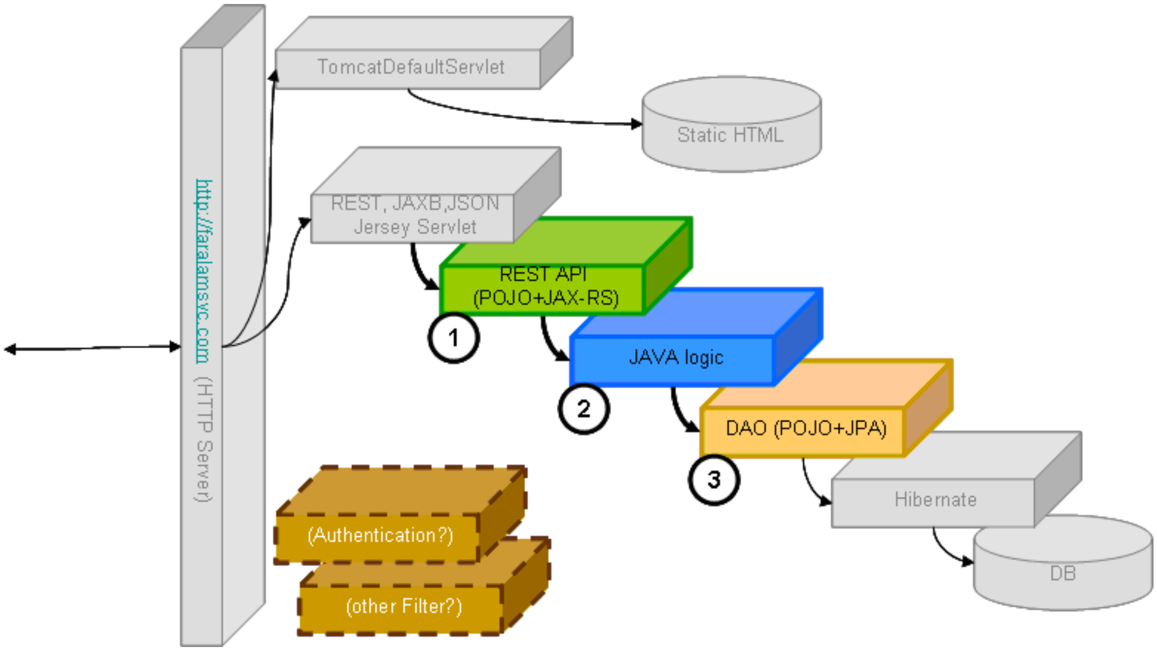


Figure 1 Technology Stack of Chalkboards. Grayed-out blocks are off-the-shelf components.

Finally, for communication we use an industry standard JAX-RS implementation (e.g. Jersey) which allows communication to the logic layer via REST/JSON standard APIs.

The entire system is deployed in a Java Servlet Container Server (e.g. Tomcat) which provides both HTTP Server functionality via the Default Tomcat Servlet as well as a multitude of other services such as Login interception (Servlet Filter) etc. The JAX-RS implementation (Jersey) provides its own interceptor Servlet which takes care of translating HTTP request responses of the type 'application/json' into standard method calls to appropriately annotated Java POJOs.

Relational Schema

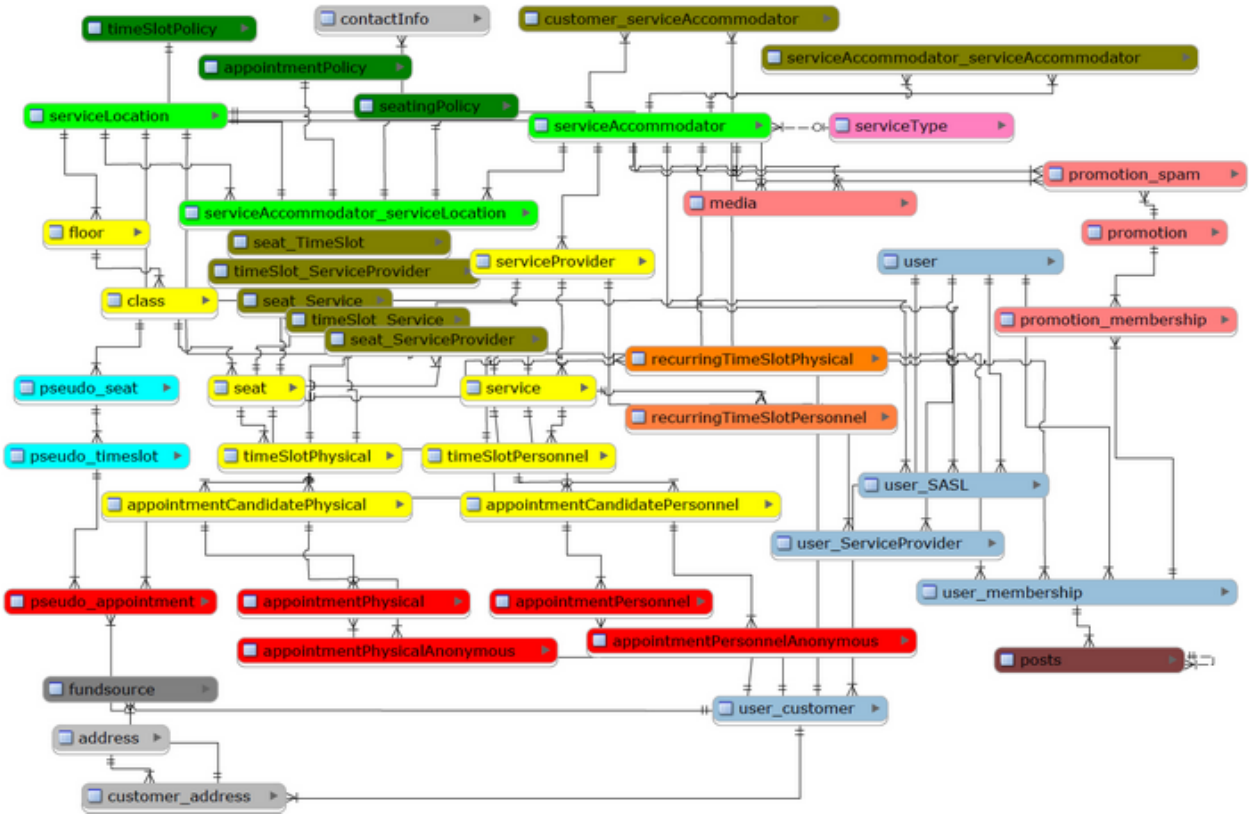
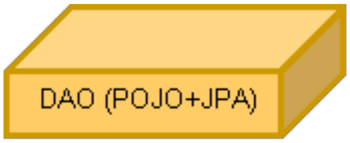


Figure 2 Schema of the Chalkboards database. Colors indicate functional groupings for ease of understanding only.

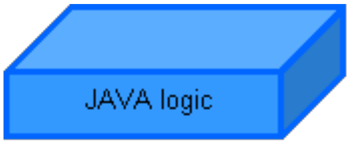
POJOS+JPA



We are currently using the Hibernate JPA implementation. Our code is completely JPA compliant (as far as the table to object mappings are concerned) so we do not need to know anything specific to Hibernate.

Every Table in the Relational Database has a corresponding Java Class along with any embedded objects as necessary. These Java Classes form the lowest level of data for the system and eventually all functionality must be represented in terms of these classes.

HIBERNATE AND JPA ANNOTATIONS



[TBD] add a few examples, example why we can’t use the auto-increment in primary key when it contains a foreign key in hibernate, etc.

JAVA LOGIC

The Java Logic layer will be in two parts. We define all relational logic as the 'Core API'. This layer is agnostic of business or domain specific meta-data. The Core API interacts with the Database and is responsible for implementing the Data Access Objects (dao). These objects are JPA2 annotated. Above this layer is the 'Domain API' layer. The Domain API is a generic term we use to define the various modules which sit above the Core API layer, and implement business logic and meta-data for various Industry Domains. Domain API is accessible externally via either the REST interface or via direct Servlet URL mapping.

Next we describe these two layers, the Core API and the Domain API, in some detail.

Core API

The first part is the Core API layer. Core API as opposed to the Domain API and the external the REST API described next. The Core API is not the outward facing API and it will attempt to remain 'generic' or industry agnostic. For example, we will try to avoid something like a 'makeRestaurantReservation' but instead have 'createAppointment' which is more generic. The main motivation for maintaining this layer of Core API is to assure that we can separate the Industry specific specializations we make to our system. The Core API will therefore match the Relational Model. The CoreAPI will recognize such artifacts as a 'Simple' service location, which has only one floor and one class; however, it will remain agnostic to anything that is specifically different for a particular Industry. The CoreAPI will therefore implement the 'generic' part of the 'degeneration' of the complex relational model into the more simplified industry specific models, but do so in parametric form rather than industry specific form.

Domain (Industry) API

The Domain API will come in many versions, as many Industries as we attempt to serve. For example, if we create a service for the Restaurant Industry, we will call this the RestaurantAPI. The methods in this API may be further specialized to suite types of restaurants, or even individual restaurants, if need be.

The [Industry]API will make no attempt to be true to the Relational Model. On the contrary, the [Industry]API will strive to match the needs of the Industry as well as the User Interface. Depending on the type of the UI (Android, IOS, Browser) we may further specialize the [Industry]API into [Platform][Industry]API segments.

The [Industry]API will in turn use the CoreAPI for any persistence related functions.

REST (JAX-RS)

This section describes the mechanisms of the JAX-RS web-service implementation. This is not a detailed API document, which will be described elsewhere.

The Theory

HTTP protocol allows HTTP Requests and responses to be exchanged between an

HTTP Server and an HTTP Client. A Web Service piggy-backs on the HTTP Request/Response mechanism to implement a remote API. The Request/Response payload in effect carries the parameters for method invocation and the results back.

REST (Representational State Transfer) utilizes the HTTP Verbs GET, PUT, and nothing else. It also requires the API to be designed in such a way that at any moment the stateless nature of the HTTP protocol is not compromised and clients therefore are not really making synchronous blocking calls as would be the case with typical remote method invocations such as CORBA etc., but rather the client is requesting the current state of a resource and requesting a change in that state, knowing that the request may or may not be successful. When an API satisfies such conditions, we call the resulting Web Service a truly RESTful service. The Chalkboards will attempt to be RESTful, but this is not a priority for us. In many cases we will assume that an invocation may effectively be a synchronous remote method invocation.

JAXB AND JAX-RS ANNOTATIONS

We will use the Jersey JAX_RS implementation for now. Details may be found on appropriate Oracle web sites (add link here).

JAX-RS REST requires two separate functionalities. First the http requests (say, GET http:// faralamsvc.com /addresses/json) must be mapped into a method call on some Java class (like Browser::getAddresses()) which will return an array of Address objects. This mapping is achieved by annotating the Browser.java class with JAX-RS path annotations (like @Path("/")) etc. Parameters from the http query are mapped as arguments to the function. The parameters can be via a query string or a POST body as appropriate.

The second functionality of the JAX-RS REST implementation requires converting the results of a Java method call above back into an HTTP response. Here the first thing to note is the mime-type of the response, which is either "application/xml' or 'application/json'. The REST implementation (e.g. Jersey) will convert the Java response into a serialized form of text as the body of an HTTP response.

The following code snippet shows how we annotate Java methods/classes in JAX-RS.

```
@GET

@Path("/addresses/xml")

@Produces("application/xml")

public List<Address> getAddresses() {

        addresses = retrieveAddresses();

        return addresses;

}
```

Our implementation will usually exchange data structures that may or not be persisted. In other words, the Objects that the UI needs may be composite objects or incomplete objects that will be matched or completed via the Java Logic layer before persistence. For example, if a restaurant has only one branch, one floor, one class etc., we may not exchange these details in the REST Web Service APIs. Instead we may create a SimpleRestaurantAPI which will strip off extra information on responses, and pre-fill the known parameters (the single floor class etc.)

on request.

This API provides access to the core functionality of the Chalkboards system. This API is 'application' agnostic. In other words, it is not aware of the type of industry for which the API is being used. The terminology for this API is therefore going to be in terms of the relation-model (see Appendix) and not in terms of the actual industry, such as the restaurant industry. This API will either use Objects from the DAO level collections of DAO objects only.

Examples of JPA and JAXB

RELATIONAL MODEL (JPA) EXCERPTS

Below we show a small part of our relational schema. The corresponding JPA annotated DAO classes (only excerpts) are shown after that. This allows us to relate how the relational model and the JPA annotations match up.

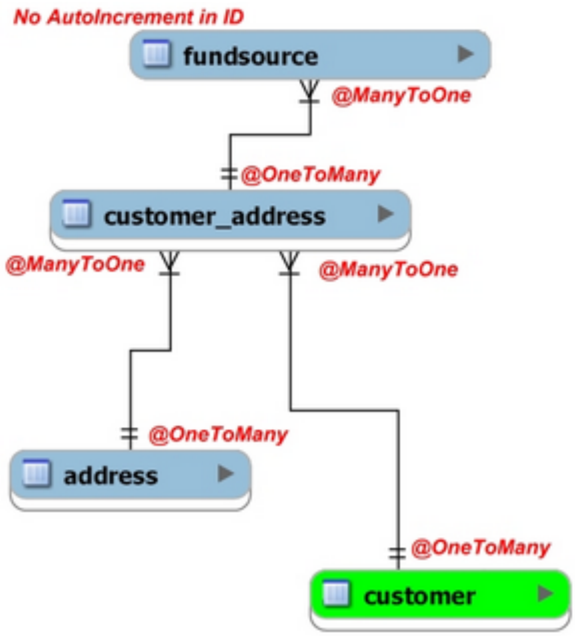


Figure 3 Relational Model. Note that 'fundsource' cannot have Auto Increment ID. A primary key that has a foreign key in it is not allowed to have autoincrement id by Hibernate, although it is allowed in SQL.

JPA Annotation excerpts

The code snippets below are from the DAO classes corresponding to the relational model above.

@Entity

@Table(name = "customer")

public class Customer implements Serializable {

```
        @Id
        @Column(name = "CUSTOMER_ID", unique = true, nullable = false)
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private int idCustomer;
        @OneToMany(fetch = FetchType.LAZY, mappedBy = "pk.someCustomer", cascade = CascadeType.ALL)
private Set<CustomerAddress> customerAddresses
}
@Entity
@Table(name = "address")
public class Address implements Serializable {
        @Id
        @Column(name = "ADDRESS_ID", unique = true, nullable = false)
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private int idAddress;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "pk.someAddress" ,cascade=CascadeType.ALL)
private Set<CustomerAddress> customerAddresses
}

@Embeddable
public class CustomerAddressPK implements Serializable {
        @ManyToOne
@JoinColumn(name = "CUSTOMER_ID_FK", referencedColumnName = "CUSTOMER_ID")
        private Customer someCustomer;
        @ManyToOne
@JoinColumn(name = "ADDRESS_ID_FK", referencedColumnName = "ADDRESS_ID")
```

```
        private Address someAddress;
}
@Entity
@Table(name = "customer_address ")
public class CustomerAddress implements Serializable {
    @EmbeddedId
    private CustomerAddressPK pk = new CustomerAddressPK();
        @Transient
        public Address getAddressh() {
                return getPk().getSomeAddress();
        }
@Transient
        public Customer getCustomerh() {
                return getPk().getSomeCustomer();
        }
        @OneToMany(fetch=FetchType.LAZY, mappedBy = "id.customerAddress")
        private Set<Fundsource> fundsources = new HashSet<Fundsource>(0);
}
@Embeddable
public class FundSourcePK implements Serializable {
        @Column(name = "ACCOUNT_NUMBER", unique = true, nullable = false)
        public String getAccountNumber() {
                return accountNumber;
        }
        @ManyToOne(fetch = FetchType.LAZY)
        @JoinColumns({
```

```
@JoinColumn(name = "CUSTOMER_ID_FK_FK", referencedColumnName = "CUSTOMER_ID_FK"),
@JoinColumn(name = "ADDRESS_ID_FK_FK", referencedColumnName = "ADDRESS_ID_FK") })
        private CustomerAddress customerAddress;
}
@Entity
@Table(name="fundsource")
public class FundSource implements Serializable {
        @EmbeddedId
        private FundSourcePK fundSourcePK = new FundSourcePK();
        public CustomerAddress getCustomerAddress() {
                return getFundSourcePK().getCustomerAddress();
        }

        public String getAccountNumber() {
                return getFundSourcePK().getAccountNumber();
        }}
```

EXCEPTIONS

PANICEXCEPTION

A PanicException is one of the two exceptions that is thrown or propagated in the Chalkboards system. A PanicException implies an error in the service itself. It means that service has encountered a logic error, missing resource, connection issue etc. Generally a PanicException usually requires human intervention. A PanicException is never thrown for any input error or violation of API contract by the caller. A PanicException is intended to inform the caller of the server error and implies that any further interaction with the server, even when not throwing an exception, should not be considered reliable.

UNABLETOCOMPLYEXCEPTION

A UnableToComplyException is the other of the two exceptions that are thrown or propagated in the Chalkboards. A UnableToComplyException is never thrown due to server internal error, resource issue or logic error. It is only thrown due to a violation of the API contract and is intended to inform the caller of such violation. A

UnableToComplyException implies that all interaction with the server is still completely reliable.

DEVELOPER DETAILS

In this section we described the steps involved in developing APIs in Chalkboards. Due to the complexity of the overall system, it is imperative that we divide the process into distinct steps which are more manageable.

The first step is to understand the APIExplorer, as it is an intrinsic part of the API development process.

Database Editing

We maintain our database schema as a MySQLWorkbench project.

API EXPLORER

(NOTE: This will be phased out in favor of using ‘Postman’ for API exploration and testing)

The API Explorer is an HTML5 web-app with hard-coded server’s names. Upon ‘Refresh’, the APIExplorer loads the API descriptions from each server. Using this description, we can run Unit Tests as a TestSuite or invoke individual APIs.

The following figure shows how the Domain unit tests (JUnit TestSuite) are executed. Note that ALL these unit tests are run via the APIExplorer Servlet. The APIExplorer runs them on the server and returns results. On the other hand APIs are invoked directly to the Server.

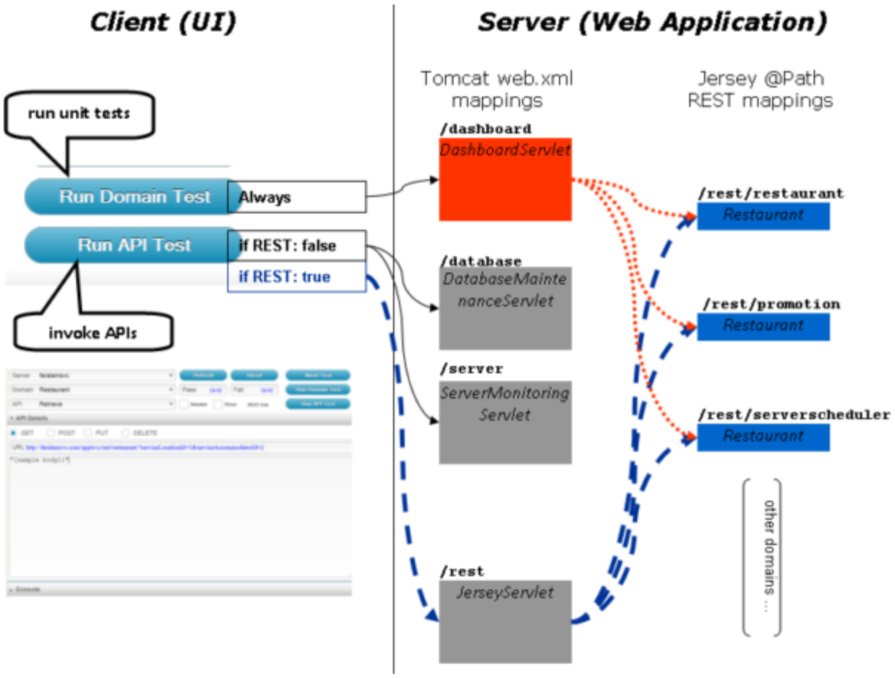
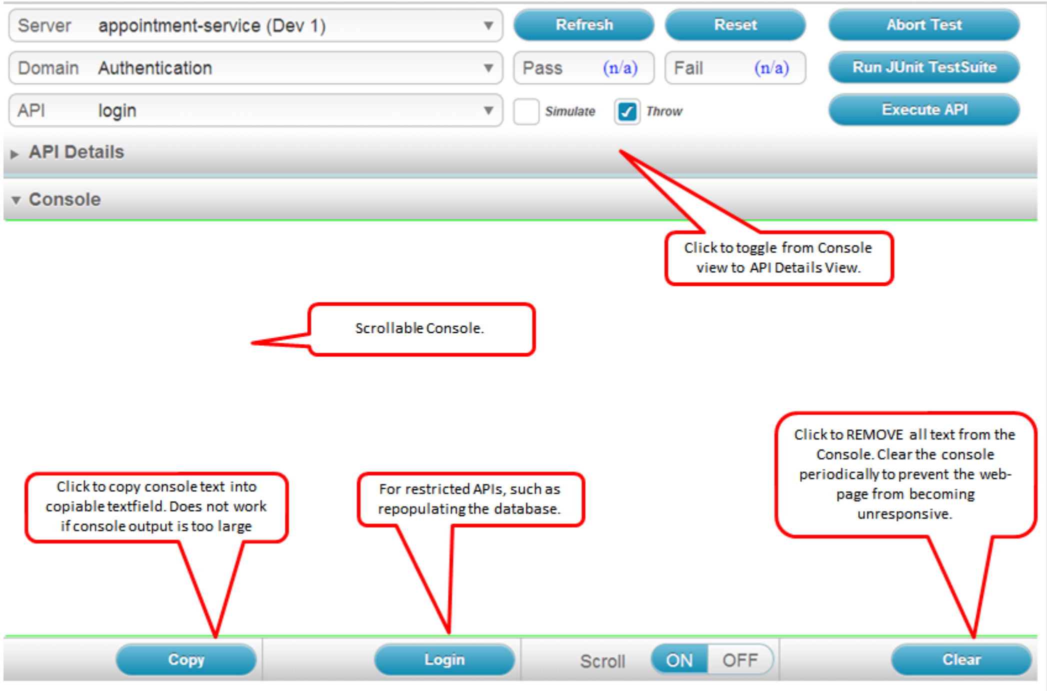


Figure 4 APIExplorer UI makes invocations to the server either to specific Servlet or to REST APIs via Jersey Servlet

The ‘APIExplorer’ firsts loads server configurations when the Refresh button is clicked. The refresh button can be clicked at any time, to retrieve the current configurations of the servers. If a server is not responding, it will be indicated via an ‘N/A’ next to the server name.

Server names/addresses are hard-coded in the APIExplorer. When a server is added or deleted, the original source code of the APIExplorer must be modified and the new version loaded to the HTTP Servers.



Click to toggle from Console view to API Details View.

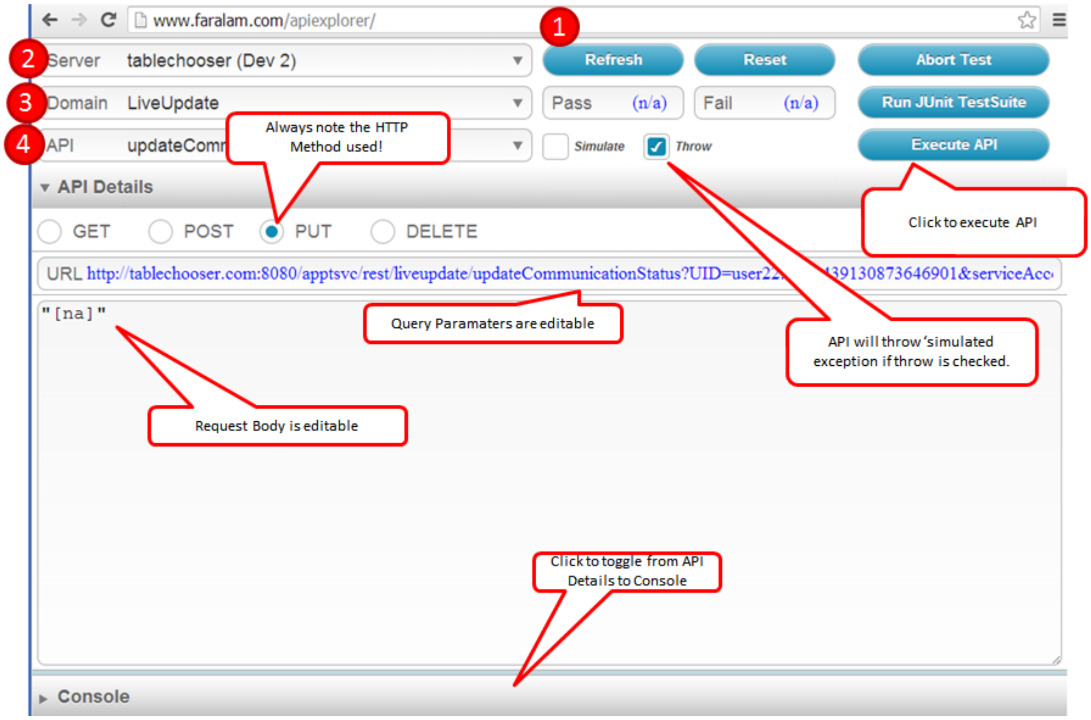
Scrollable Console.

Click to REMOVE all text from the Console. Clear the console periodically to prevent the web-page from becoming unresponsive.

Click to copy console text into copiable textfield. Does not work if console output is too large

For restricted APIs, such as repopulating the database.

Figure 5 APIExplorer screen with console visible.



URL http://tablechooser.com:8080/apptsvc/rest/liveupdate/updateCommunicationStatus?UID=user22 ... 39130873646901&serviceAcc

"[na]"

Always note the HTTP Method used!

Click to execute API

Query Paramaters are editable

API will throw ‘simulated exception if throw is checked.

Request Body is editable

Click to toggle from API Details to Console

Figure 6 APIExplorer showing API Details view expanded. NOTE: All changes will be lost if user clicks on any other API, and then comes back to the current API.

Steps in Using the APIExplorer

The following steps are the typical usage pattern of the APIExplorer. Upon loading the APIExplorer, go through the following steps.

1. ***Refresh***: (Step 1) This button makes the app attempt to retrieve server configurations. If a server does not respond, and 'N/A' is pre-pended to its name. Otherwise, we populate the drop downs with the details for the server.

2. ***Select Server***: (Step 2) We have multiple servers. 'localhost:8080' is also included, but it points to your local machine and you must have tomcat running. This list of servers is hardcoded into the app.

3. ***Select Domain and run TestSuite***: (Step 3) Each server supports multiple domains. A 'domain' is a loose term we use to group functionality. For example, the core Chalkboards functions are called 'apptsvc' domain. While the Restaurant services, which sit on top of this, are called 'restaurant' domain. Each domain has a bunch of unit tests, which we call the TestSuite.

4. ***Invoke API***: (Step 4) Each domain implements various APIs which are exposed via the REST mappings. When the APIExplorer loads the server configuration, it knows these path mappings. Clicking on any API pre-fills the URL, GET/POST… , body, etc., with sample values. You can modify these values before clicking 'Run API'.

Run Unit Tests (TestSuites)

The 'Run TestSuite' button will execute this unit tests on the domain selected on the server and report back the results. Note that it will run the entire battery of Unit Tests. The tests are run on the server side and for each test we get a summary result back (PASSED/FAILED etc.).

Invoke API

Clicking on the 'Run API Test' invokes the API. Its takes the values from the 'API Details' section, so any changes you make, will be the ones used. Two additional buttons are provided for API testing.

Simulate: If checked, we add the simulate=true argument to the API invocation. This results in a canned response and is useful for development and debugging.

Throw: If checked, the API will throw a simulated exception. This helps us verify our error handling code on the client side.

Authoring for API Explorer

Next we described how the domain and the API details are described. This is the description that the APIExplorer loads when the 'refresh' button is clicked.

Every domain has an 'apiDescription.json' file at the root of its …/test directory. This file looks as follows:

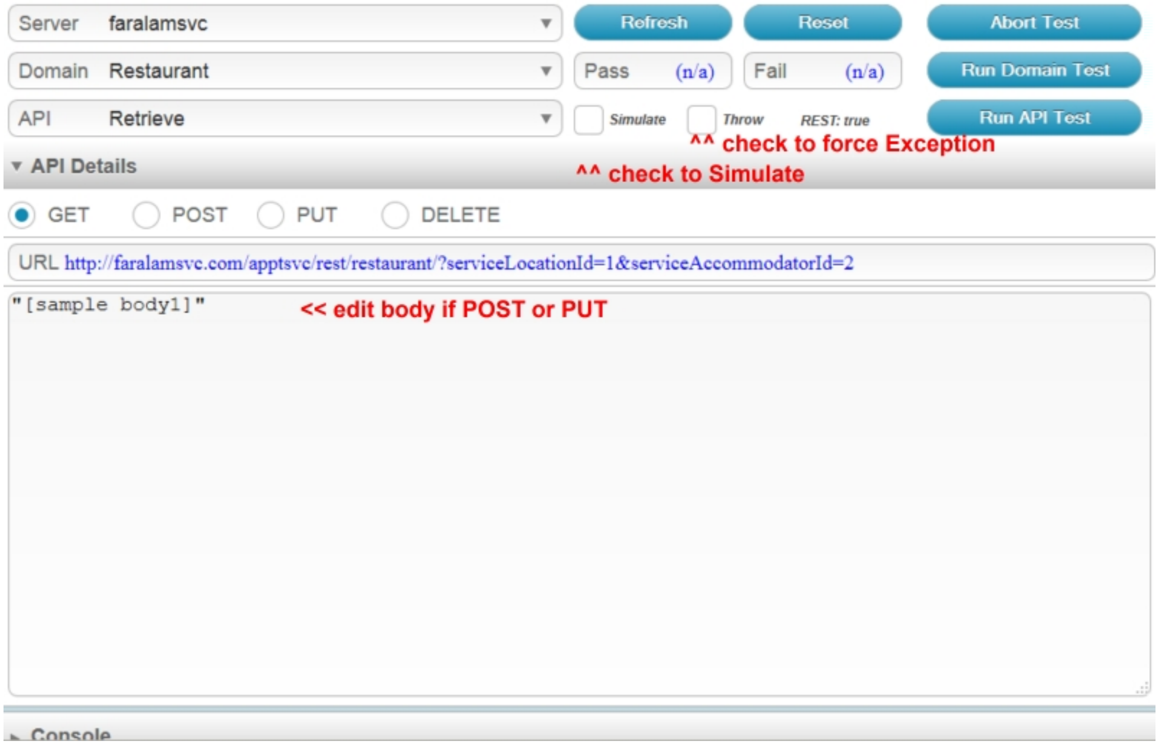
com/faralam/domains/restaurant/test/apiDescription.json

```
{"value":"Restaurant","urlSuffix":"/restaurant", "sampleQuery":"?testIndex=2",

 "description":"Suite of low level tests in Chalkboards core",

 "api":[

    {"value":"Retrieve", "urlSuffix":"/rest/restaurant","type":"GET",

      "sampleQuery":"/servicelocation/1/serviceaccommodator/1",

      "sampleBody":"[sample body1]",

      "description":"Retrieve Restaurant"},
    {
      <add new API description here>

    }

    ]

}
```

Every domain has an 'apiDescription.json' file at the root of its …/test directory. This file as above. This is in JSON format and will be parsed by the client. The APIExplorer retrieves one such file per domain.

The first part describes the domain itself (value is the name), and how to run the unit test. The total test count is added dynamically to this file.

The sample file has only ONE API described. We know it must be invoked with GET, we know its path and sample arguments.



Server faralamsvc ▾

Domain Restaurant ▾

API Retrieve ▾

Pass (n/a) Fail (n/a)

□ Simulate □ Throw REST: true

^^ check to force Exception

^^ check to Simulate

● GET ○ POST ○ PUT ○ DELETE

URL http://faralamsvc.com/apptsvc/rest/restaurant/?serviceLocationId=1&serviceAccommodatorId=2

```
"[sample body1]"
```

<< edit body if POST or PUT

Figure 7 API Details in APIExplorer for apiDescription.json example. Note how the final URL is the concatenation of the base URL and the urlSuffix from the description file. For Simulate flag, simulate=true will be appended to the URL.

To add a new API, one must add a new section to this document. To add a new JUnit test, only the *TestEnumerator.java file has to be updated in the appropriate domain test directory. For example, for the Restaurant domain the RestaurantTestEnumerator.java file is in the directory com/faralam/domains/restaurant/test.

Knowledge Review for Developers

Answer the following questions.

1) Why do we always throw checked exceptions in our server? Why don't we throw un-checked or runtime exceptions? (Hint: What does it mean to say that an unchecked exception shows up in unexpected places, where it is out-of-context.).

2) What are the two exceptions that are thrown by all methods in the server? What is the difference between them? (Hint: Either the server encountered an error, or the user made an inconsistent request…)

3) Why don't we marshal 'hibernate' objects (Entities managed by Hibernate … i.e. objects annotated with JPA annotations) out directly? Why do we always convert them to JAXB annotated objects? What would be the problem if we tried to marshal a Hibernate Managed object?

4) How do we propagate an exception from the server to the client (say, IPhone) via REST/JSON? (Hint: http codes in conjunction with json message body could emulate our two exceptions. What error codes would

fit which exception?)

5) In Hibernate, for two-way mappings, (parent->child and child<parent) why do we always have one side as passive? Why don't we need this in Java objects when they have parent-child relationships? (Hint:

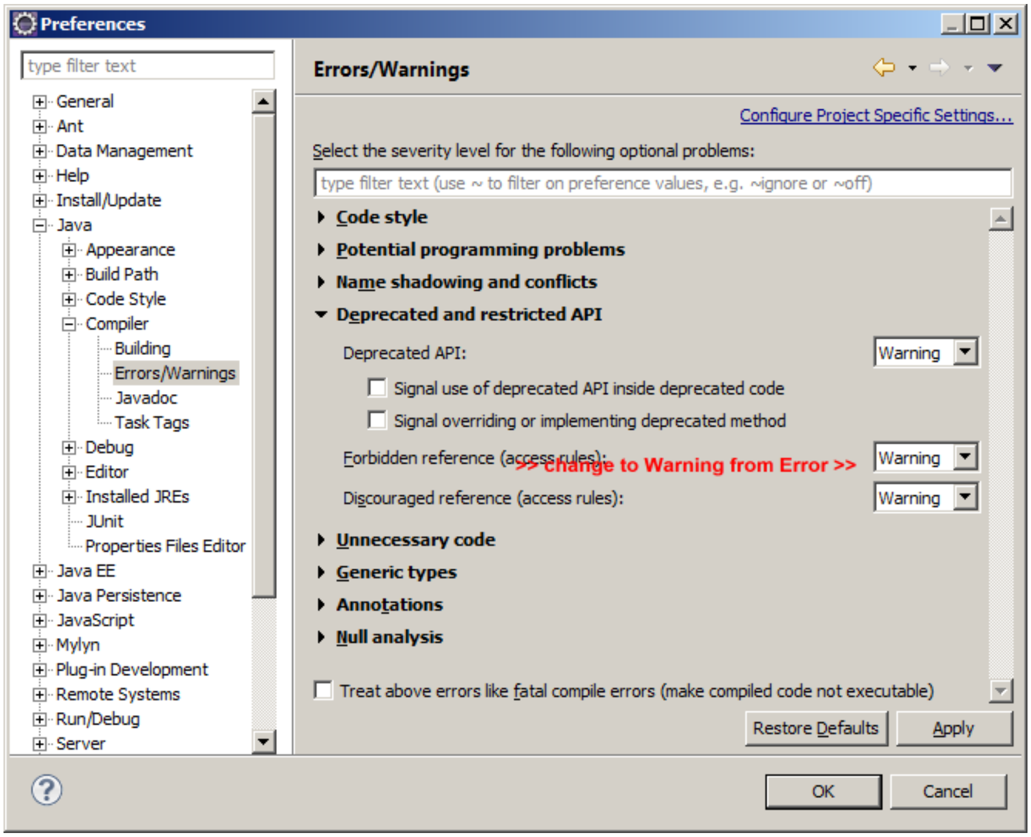
Server Synchronization

Become familiar with our Amazon EC2 server. Basically this is a Linux box running tomcat, MySQL etc.

Our ant build file builds a war file which we then upload to our servers. Details to be added.

IDE and Dev Environment

We use Eclipse, Filezilla, Putty. Our project is maintained as an eclipse project.



Select the severity level for the following optional problems:

type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)

Deprecated API: Warning

□ Signal use of deprecated API inside deprecated code

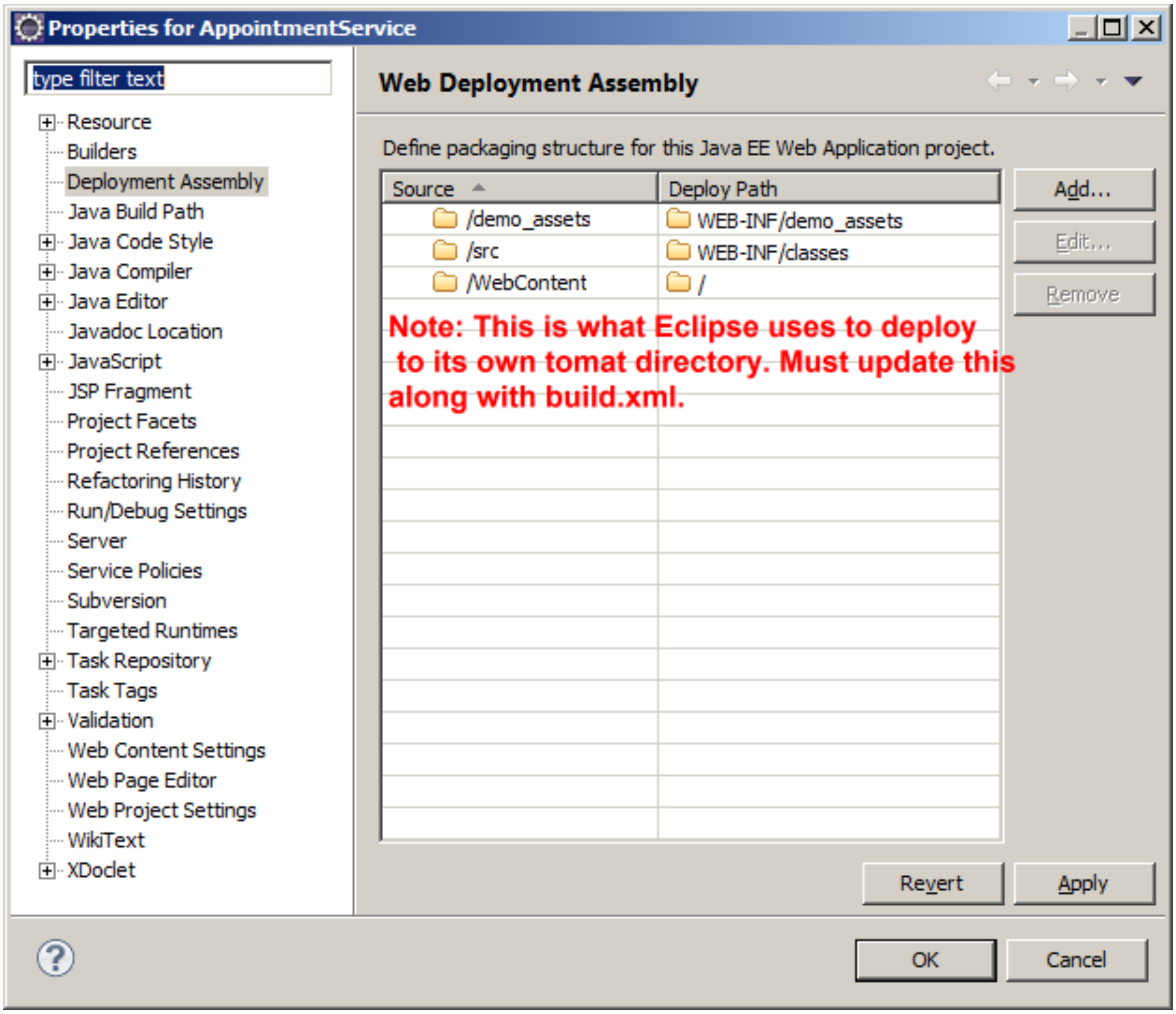
□ Signal overriding or implementing deprecated method

Forbidden reference (access rules): Warning >> change to Warning from Error >>

Discouraged reference (access rules): Warning

□ Treat above errors like fatal compile errors (make compiled code not executable)

Figure 8 You may have to disable the warning for restricted api in order to build.



Web Deployment Assembly

Define packaging structure for this Java EE Web Application project.

Source	Deploy Path
/demo_assets	WEB-INF/demo_assets
/src	WEB-INF/classes
/WebContent	/

Note: This is what Eclipse uses to deploy to its own tomat directory. Must update this along with build.xml.

Figure 9 Eclipse uses its own idea of what should be deployed. Eclipse does not use the WAR file we create. So we need to update both build.xml and this window when we add a directory. In this case the directory is ‘demo_assets’. Currently we use ‘restaurant’.

Steps in developing an API

These are the common steps that need to be followed for developing a new API.

1. Determine if any new Java Types are required. Remember that we cannot marshal primitive types via the REST webservice. Therefore, if we need to return an integer, for example, we have created an object and wrap the integer in a name-value pair.

2. Implement the types. This may involve annotation the object using the JAXB annotations. The JAXB annotations are used by the marshaling code in the REST servlet to convert the java object to json.

3. Implement the API method in the domain. For example, if the API is to be in a domain service, says Restaurant, then the method will be implemented in the RestaurantService.java class. The method must

implement the signature completely, -input and output (return) values/types must be implemented as required by the use case. However, at this point, the method should only work for input parameter 'simulate'=true. It should throw an exception (PanicException) indicating that the method is not implemented yet. For simulate=true, it should take in parameters and return dummy/fake data.

4. Update the apidescription.json for this method, as described earlier in the section on apiexplorer. The objective it to allow our apiexplorer web-app to be able to discover this new method.
5. Deploy the server locally (war on localhost tomcat). Now, if we point our apiexplorer web-app to 'localhost' as the server, we will be able to test the new API. If the apidescription modification was correct, we should see the entry for the new method in the apiexplorer.
6. Invoke the API from the apiexplorer, make sure you have simulate checkbox checked. You should see the fake data.
7. Implement a Unit Test. This will execute the method. However, now we have to make sure that we implement the real logic of the method. Therefore, now we have to access the database (via Chalkboards) and therefore, we may need to decide if we have to modify the schema or if we need 'adapter' classes to convert between JAXB annotated classes to the JPA annotated classes inside the server. We also have to implement the logic for real database access. Basically, we are fleshing out the method for real now. We should now be able to run this unit test directly (via eclipse, and JUnit).
8. Finally, we build and deploy the new war file to the local host.
9. Re-Deploy the server locally (war on localhost tomcat). We should now be able to invoke the API without setting the simulate flag. (by default simulate=false). Now we can deploy to one of the external servers for final testing (via APIExplorer).

These steps, tedious as they may look, actually should prevent bugs/errors, which are even more tedious to fix

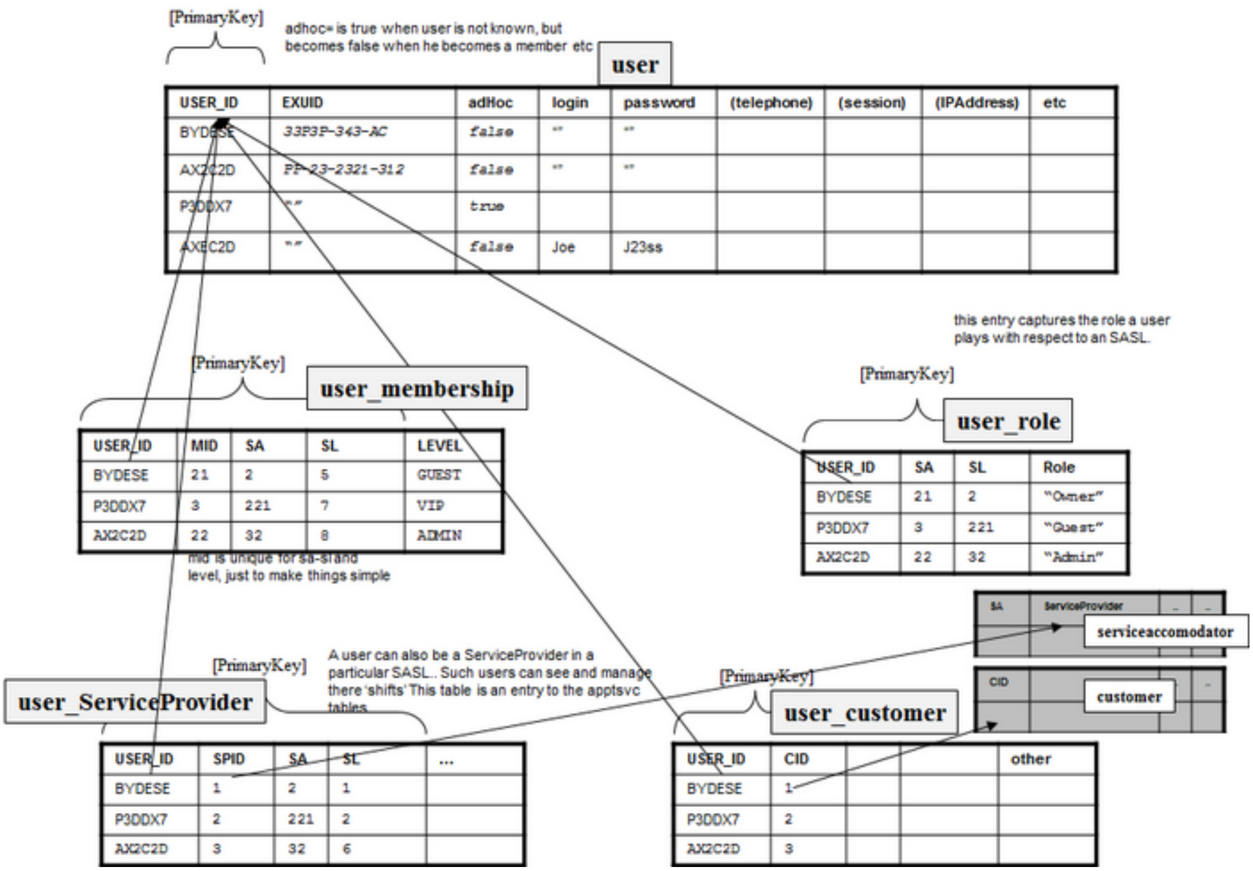
AUTHENTICATION/USER MANAGEMENT DETAILS

Chalkboards supports multiple authentication schemes. The entire system is based on UIDs, which form the key into the system. All external authentications must be mapped to a unique internal USER_ID.

The USER_ID, a Unified User Identity

The objective is to provide a unified user-management and authentication system such that on the one hand we can identify all users uniquely, but on the other hand, we can distinguish the various different roles/capabilities or identities they may take in different parts of the system. To this end, we declare a universally unique identifier, the USER_ID that identifies any user throughout our system.

We implement the various associations/roles/capabilities of the user via separate relational tables.


[PrimaryKey]
adhoc= is true when user is not known, but becomes false when he becomes a member etc
user

USER_ID	EXUID	adHoc	login	password	(telephone)	(session)	(IPAddress)	etc
BYDESE	*33P3P-343-AC*	*false*	“”	“”				
AX2C2D	*PP-23-2321-312*	*false*	“”	“”				
P3DDX7	“”	true						
AXEC2D	“”	*false*	Joe	J23ss				

[PrimaryKey]
user_membership

USER_ID	MID	SA	SL	LEVEL
BYDESE	21	2	5	GUEST
P3DDX7	3	221	7	VIP
AX2C2D	22	32	8	ADMIN

mid is unique for sa-sl and level, just to make things simple
this entry captures the role a user plays with respect to an SASL.
[PrimaryKey]
user_role

USER_ID	SA	SL	Role
BYDESE	21	2	"Owner"
P3DDX7	3	221	"Guest"
AX2C2D	22	32	"Admin"

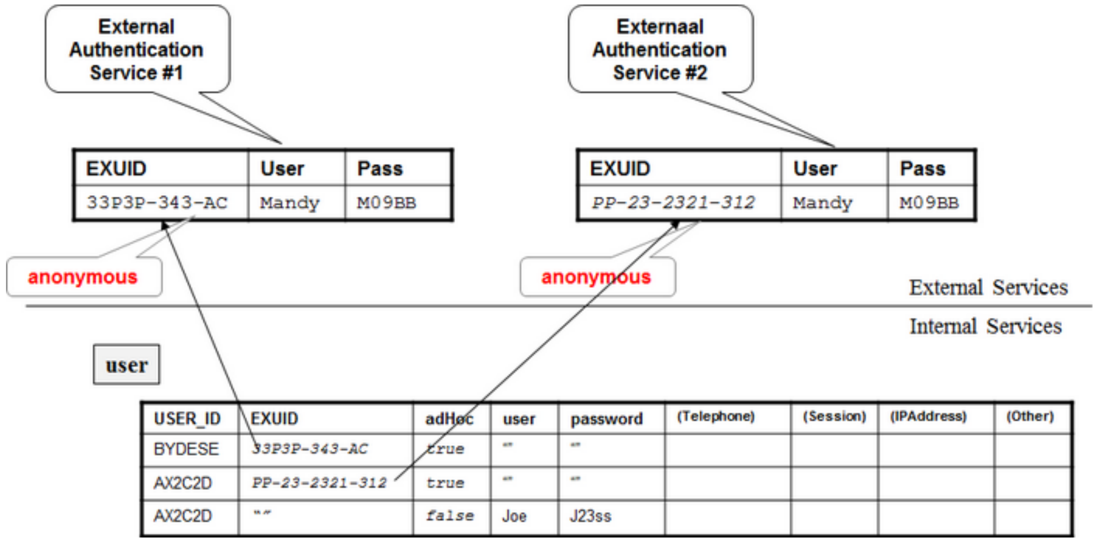
SA
ServiceProvider
serviceaccomodator
CID
customer
[PrimaryKey]
A user can also be a ServiceProvider in a particular SASL.. Such users can see and manage there 'shifts' This table is an entry to the apptsvc tables
user_ServiceProvider

USER_ID	SPID	SA	SL	...
BYDESE	1	2	1	
P3DDX7	2	221	2	
AX2C2D	3	32	6	

[PrimaryKey]
user_customer

USER_ID	CID			other
BYDESE	1			
P3DDX7	2			
AX2C2D	3			

Figure 10 User management relations


External Authentication Service #1
Externaal Authentication Service #2

EXUID	User	Pass
33P3P-343-AC	Mandy	M09BB

EXUID	User	Pass
PP-23-2321-312	Mandy	M09BB

anonymous
anonymous
External Services
Internal Services
user

USER_ID	EXUID	adHoc	user	password	(Telephone)	(Session)	(IPAddress)	(Other)
BYDESE	*33P3P-343-AC*	*true*	“”	“”				
AX2C2D	*PP-23-2321-312*	*true*	“”	“”				
AX2C2D	“”	*false*	Joe	J23ss				

Figure 11 External authentication. Could be via Facebook or some other customer user-authentication scheme. NOTE that username is empty in our tables.

Authentication (HTTP/REST)

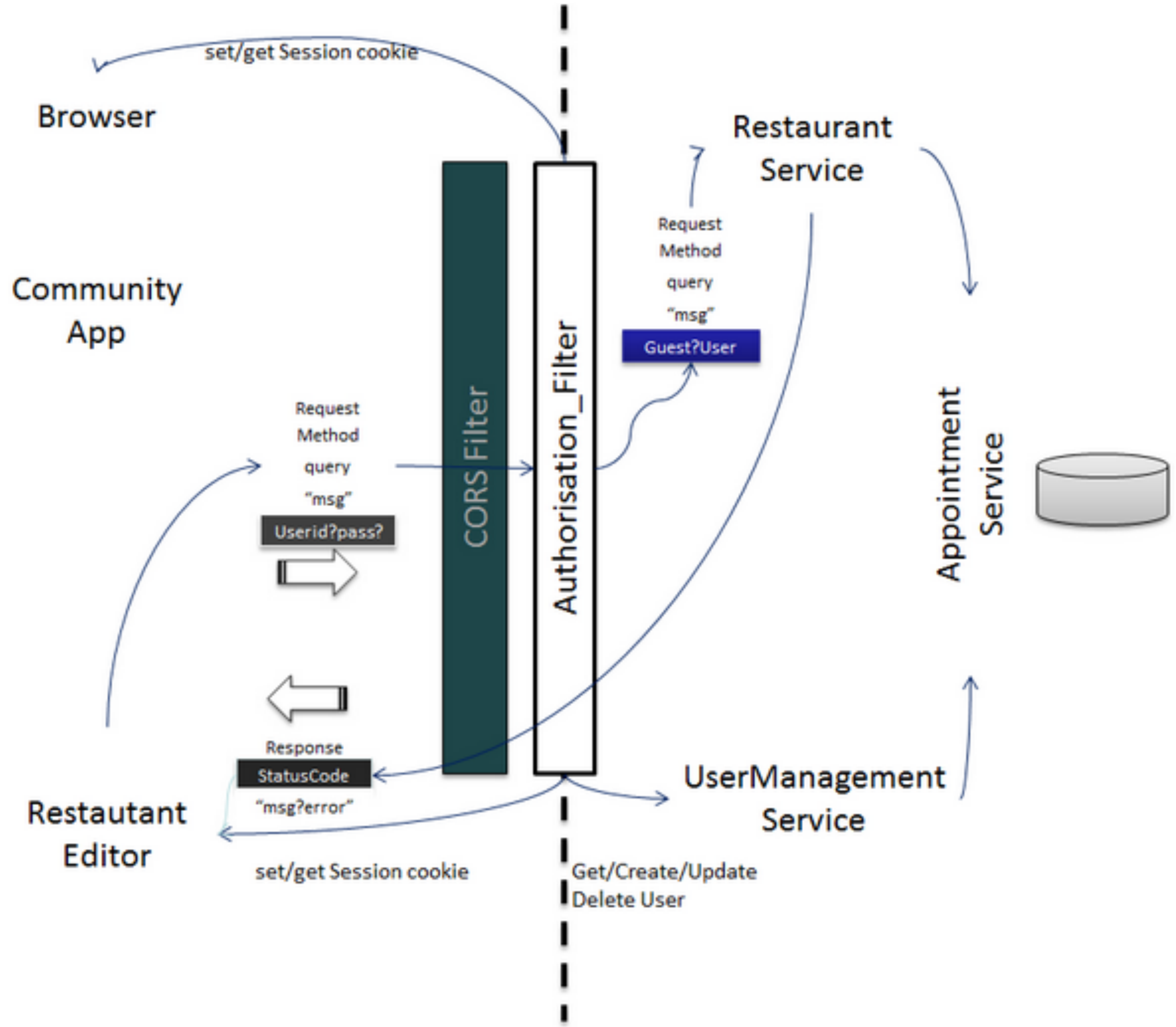


Figure 12 Basic flow for Rest Authentication filter using basic authentication.

AUTHENTICATION (OAUTH)

[tbd] Describe how we use the Facebook etc. redirection to allow external authentication.

AUTHENTICATION CHALKBOARDS (API)

The service allows a App or Portal to call 'getUser' by sending a bunch of parameters in order to verify if the user is logged in currently or recognized. Either way, a User object is returned that contains the USER_ID. Other methods are login, logout. Login called with POST will create a new user.

User CRUD methods

createUser

REST Mapping:	*[POST] /authentication/user*
Description:	*Create a new User with given username and password*
Parameters:	*username, password,*
Return Value:	*User object*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*createUser (username,password, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

updateUser

REST Mapping:	*[PUT] /authentication/user*
Description:	*update an existing user with the new user object. Userid cannot change.*
Parameters:	*user object (modified). userid must remain same.*
Return Value:	*User object*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*deleteUser (user SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

deleteUser

REST Mapping:	*[DELETE] /authentication/user*
Description:	*delete an existing user*
Parameters:	*userid*
Return Value:	*User object or Result object.*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*

Syntax:	*deleteUser (userid, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

retreiveUser

REST Mapping:	*[GET] /authentication/user*
Description:	*retrieve a user for a given userid*
Parameters:	*userid*
Return Value:	*User object or Result object.*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*retrieveUser (userid, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

Role (user_role) CRUD methods

setRole

REST Mapping:	*[POST] /authentication/role*
Description:	*create a new role for a user in an SA/SL.*
Parameters:	*userid, sa, sl, role string.*
Return Value:	*Role object*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*setRole (userid, sa, sl, newrole, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

getRole

REST Mapping:	*[GET] /authentication/role*
Description:	*Retrieve the role for a userid, sa, sl.*
Parameters:	*userid, sa, sl,*
Return Value:	*Role object*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error. UnableToComplyException if no role exists.*
Syntax:	*getRole (userid, sa, sl, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

updateRole

REST Mapping:	*[PUT] /authentication/role*
Description:	*Change the role string of a userid,sa,sl.*
Parameters:	*userid, sa, sl, role string*
Return Value:	*Role object*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*updateRole (userid, sa,sl,role, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

removeRole

REST Mapping:	*[DELETE] /authentication/role*
Description:	*Remove the role string of a userid,sa,sl.*
Parameters:	*userid, sa, sl*
Return Value:	*Role object*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*removeRole (userid, sa,sl,, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	

Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

API specific methods (user, role etc)

NOTE: getUser is a convenience method for the clients, …. it is different from retreiveUser above. getUser takes a bunch of parameters and tries to ‘guess’ who the user is. it never takes the userid.

getUser

REST Mapping:	*[GET] /authentication/getUser*
Description:	*Send a whole bunch of parameters retrieved from the client (currently none), to the service to verify if the user is recognized or not.*
Parameters:	*NONE*
Return Value:	*User Object. This object contains the USER_ID to be used for all future calls, as well as other values. These objects describe the relationships the User has with the Community System. In case the user is anonymous or ad-hoc, the other fields will be empty.* ***This object has a field that tells the App if the user is LoggedIn*** *or not (similar to session). In most cases we will remember the user and login.*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*getUser (NONE, SecurityContext)*
Core API Counterpart:	*TBD*
Data Access Objects:	*TBD*
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

login

REST Mapping:	*[GET] /authentication/login*
Description:	*Send userid and password to login user.*
Parameters:	*userid, password*
Return Value:	*User Object. This object contains the USER_ID to be used for all future calls, as well as other values. These objects describe the relationships the User has with the Community System. In case the user is anonymous or ad-hoc, the other fields will be empty.* ***This object has a field that tells the App if the user is LoggedIn*** *or not (similar to session). In most cases we will remember the user and login.*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid. If user is not recognized, this exception is thrown also.* *PanicException is thrown if there is an internal server error.*
Syntax:	*login(userid, password, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	

Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

logout

REST Mapping:	*[GET] /authentication/logout*
Description:	*Send userid to logout user*
Parameters:	*userid*
Return Value:	*RESULT object.*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*logout(userid, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

isUsernameAvailable

REST Mapping:	*[GET] /authentication/isUsernameAvailable*
Description:	*Send username to verify if it is available. Otherwise, provide suggestions*
Parameters:	*username*
Return Value:	*RESULT object. (will contain selection of alternatives if result is false*
Exceptions:	*UnableToComplyException is thrown if supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Syntax:	*isUsernameAvailable(username, SecurityContext)*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.accesscontrol.api.AuthenticationService*

TODO: Add verification methods (syntactical suger-coating) to check if user has role etc.

RESERVATION SERVICE API DETAILS

For every SASL, (an entry in the Se**rviceAccommodatorServiceLocation** table) there are several additional tables in which we store extra rules and meta-data for that sasl. We describe them next.

PSEUDO RESERVATIONS AND POLICIES

A Pseudo Reservation is defined as a reservation that is not against an actual physical seat. It is essentially a place in a queue, with the understanding that the service location can handle the people in the queue at any given time. This is also the most typical form of reservation in a restaurant. The actual seating is done dynamically, by the hostess, but at any given time (say 6pm) the restaurant could have 5 people signed up. All 5 would need a free table at 6pm, for the reservation to be real. However, due to the dynamic nature of how people eat (how long they stay, when they show up etc), the restaurant may ask someone to wait a while for a free table. In effect, when people 'make a reservation' at a restaurant, often there is no real reservation, … there name is just put in a queue for that specific hour when they made the reservation.

So how do we define time-dependent queue length in a 'Class or Tier' of an SASL?

We do so by creating an entry in the ClassTimeSlotPolicy table. The table is shown below (primary keys are summarized into single columns). Note that there is a column for a foreign key of a Promotion as well. So, a policy can be by itself or it can be associated with a promotion. This allows us to accommodate different policies based on regular seating and promotion seating.

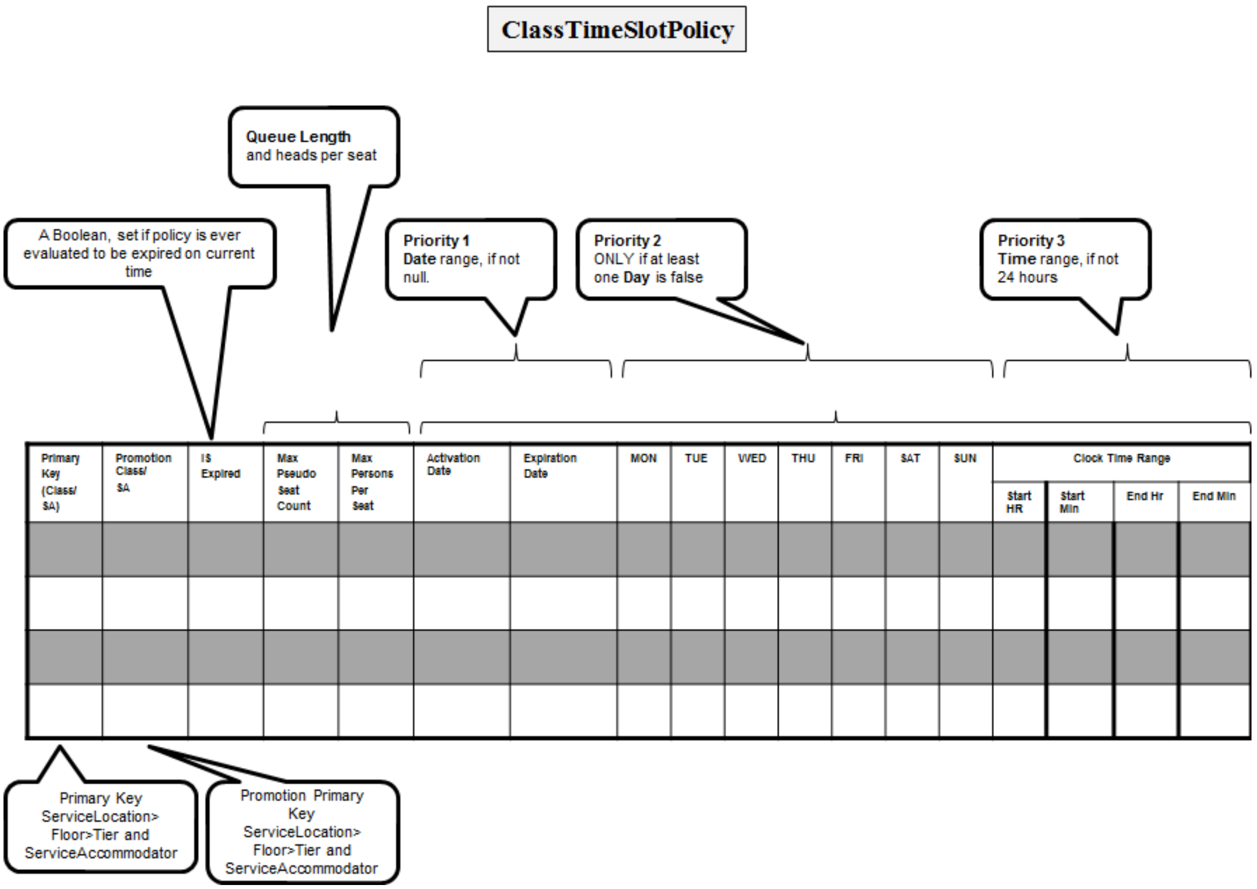

ClassTimeSlotPolicy
Queue Length and heads per seat
A Boolean, set if policy is ever evaluated to be expired on current time
Priority 1 Date range, if not null.
Priority 2 ONLY if at least one Day is false
Priority 3 Time range, if not 24 hours
Primary Key (Class/ SA)
Promotion Class/ SA
IS Expired
Max Pseudo Seat Count
Max Persons Per Seat
Activation Date
Expiration Date
MON
TUE
WED
THU
FRI
SAT
SUN
Clock Time Range
Start HR
Start Min
End Hr
End Min
Primary Key ServiceLocation> Floor>Tier and ServiceAccommodator
Promotion Primary Key ServiceLocation> Floor>Tier and ServiceAccommodator

Figure 13 ClassTimeSlotPolicy table showing different types of information. The policies must be verified in their order of priority so that we can optimize evaluation.

ClassTimeSlot Policy

The 'ClassTimeSlotPolicy' table hold policy entries that can be used to describe the queue length at various times in a 'Tier or Class' in an SASL. For example, if the Patio of a restaurant is a Tier, then the policies regarding this Tier will have the Tier as part of its foreign primary key.

The policy entries contain several bits of information.

1. IsExpired: First, we have a Boolean 'isExpired' which is false when the policy is created. However, when we evaluate a policy at the 'current' time, and find out that the policy has expired (assuming it is date-bound policy) then we set this flag to true. We can do this because time only moves forward. If it is expired today, it is expired forever. This allows us to clean up expired policies without having to evaluate them again.
2. TimeSlotType: This tells us what the length of the slots are, …. are they every fifteen minutes? Hour? Completely flexible? etc
3. QueueLength: Next are the MaxPseudoSeatCount and MaxHeadsPerSeat. Together they define the Queue

Length.

4. The rest of the columns together describe when this policy is valid. Basically, a time range can have 'dates+time', or 'days+time' or just time. It is how people usually talk about 'opening hours'. Examples of 'when a policy is valid' are in human terms sentences like the following.

    1. We are open every day 9 to 6 (just time) (so, this is true for seven days)

    2. except Mondays when we are open 9 to 4 (days+time)

    3. and on Jan 1 we open only 4-9 (date+time)

We model our system the same way, with ClassTimeSlotPolicy entries. The time ranges during which a policy is valid is captured by taking human readable rules and converting them to our format. NOTE: We capture 'Days' of validity by simply having seven Booleans, one for each day, and setting them true or false. All SEVEN days being true, however, is the same as saying 'it does not matter what day'. This is reflected in the way we prioritize the rules, described next.

Next we have the question of Priorities in the policies.

As such, each policy entry exists on its own, but taken together, some will override others.

For example, a restaurant is open every day except Monday. But on Jan 1, which is a Monday (for example), it is claiming to be open. So, we have a policy entry that Activates and Expires on Jan 1 (some year), and we have a policy entry for 6 Days. The first entry is a higher priority.

Finding the Applicable Policy

The problem of evaluating policies can be summarized as, given a time range (say Monday, Jan 1, 6 to 7PM) how many seats are available?

If we have already arranged the policies by priority, then we start evaluating them from the top. If the 'Jan' policies are true (it covers the time in question) then we look up the queue length for this policy entry, and we have our answer by now comparing this queue length with existing reservations. So it is imperative that we arrange the policies in the right order of priority (which overrides which).

There are two additional fields we have not discussed, -timeslot length and reservation type. For simplicity we assume that all the policies for a particular institution have the same length.

Finding Empty Slots

ASSUMPTION: To simplify things we make an **assumption** (for this discussion) that the **TimeSlotType for all policies in the set are HOURLY** (or every hour). In other words, the policies are written in terms of hours and the generated time-slots will be by the hour. However, the actual reservation is not restricted in this way. The reservation itself can cross hour boundaries (so a reservation can be from 8:10 to 9:33). We also assume that a generated timeslot (hourly) is considered taken if a reservation spills into it even by one minute. So, a reservation from 8:00 to 9:01 effectively spans two time-slots, 8 to 9 and 9 to 10. The overlap that can be tolerated (deltaOverlap) is considered zero.

To determine how many slots are available for a given range, we have to find out how many non-overlapped reservation slots are available. We also assume that we do our computations for one day (although this is easily changed).

In order to speed up the computation, we use an array of integers (one for every minute of the day). So, we first create an array Integer[] = new Integer[24x60]. Let us call this the ReservationsByMinuteArray.

The idea is to use this array as an accumulator as we evaluate every existing reservation, to an accuracy of 1 minute.

We compute the existing reservation situation by updating the integers in the above ReservationByMinuteArray for every day. We do this as follows: For every existing appointment, we increment the integers within that time range by one. So, if there is only one existing reservation, from 6 to 7pm, then the integer array for the day will be all zeros except for 1's from 6 to 7pm.

We now repeat this process for all reservations for that day, incrementing the integers for every appointment. If a reservation is 7:10 to 8:10, it spills over into the next hour, so the first 10 integers of the hour between 8 and 9 are incremented by one. After exhausting the list of current reservations, we are left with an array of integers with numbers indicating the reservation count by the minute. Now we can generate a list of available time slots by taking the highest number in any range (so, 8 to 9pm will have one reservation, although it is only during the first 10 minutes), and then subtracting from the queue length for that slot as determined by the policy that is valid at that time.

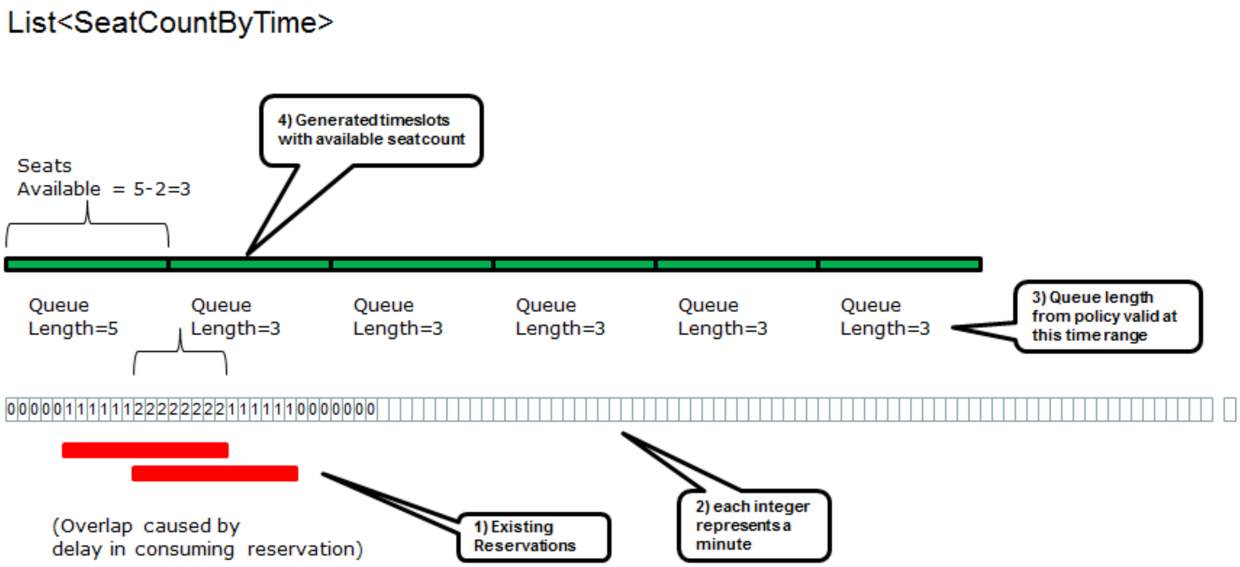

List<SeatCountByTime>
4) Generated timeslots with available seatcount
Seats Available = 5-2=3
Queue Length=5
Queue Length=3
Queue Length=3
Queue Length=3
Queue Length=3
Queue Length=3
3) Queue length from policy valid at this time range
00000111111222222221111110000000
(Overlap caused by delay in consuming reservation)
1) Existing Reservations
2) each integer represents a minute

Figure 14 We allow final reservations (in red) to be overlap timeslots, to adapt to the reality of how reservations are consumed (people came in 10 minutes late). But 'available' timeslots (green) are generated by using the fixed timeslot length determined by the policy. The seat count is determined by subtracting max in current reservation from policy-max.

This tedious process results in a series of time-slots and the available seat counts at that slot, -which satisfies the policies conservatively.

This SeatCountByTime series which is essentially a series of beginning and end times with available slots, can now be used by an App to show a customer how many seats are available.

When a reservation is made using one of these slots, no further evaluation is made, -since these are 'Pseudo Reservations', and there is no real 'Seat' associated with them. A PseudoSeat is indeed assigned, but these pseudo-seats have no physical counterpart.

However, to help keep track of where a person is actually seated finally, a PseudoReservation does have a field to hold the seat number/name/designation. This information is entered when a PseudoReservation is 'consumed'.

Type of reservation Supported

We store the type of reservations is supported at any timerange for a particular user.

We need to define a type, ReservationType.

We need to define a type, ReservationTypeRuleSet which will contain all the logic in the form or an array of TimeRange, reservation type, usertype (*membership* level).

Get the reservation type supported for a user at any given time.

REST Mapping:	*[GET] /restaurant/reservationtype*
Description:	*For the given timerange and restaurant, usertype, what type of reservation is supported? User type is the level of membership of a user in a restaurant.*
Parameters:	*Array of QueueLength, PrioritizedTimeRange, (and SecurityContext injected by Jersey)* *The queue length is provided per time range. So, 3,timeRange1 means that for timerange1 the queue length is 3.*
Return Value:	*ReservationType object. Inside we will have the value (enum) of what type of reservations are supported for this user at this time.*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	*Chalkboards.getDomainVariable*
Data Access Objects:	*com.faralam.domains.restaurants.dao.AppointmentPolicy and others*
Business Logic:	*The appointmentpolicy table will contain a ReservationTypeRuleset object in the form of a string, which will be interpreted into the ReservationTypeRuleset object in the domain layer, and then used with the current inputs to determine which type of reservation should be supported for this user at this time.*
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	*getReservationTypeSupported(SASL, usertype, TimeRange, SecurityContext)*

Set /modify the type of reservation supported. We store this in the form of a ReservationTypeRuleSet

REST Mapping:	*[PUT][POST] /restaurant/ reservationtype*
Description:	*For the given timerange and restaurant, usertype, what type of reservation is supported? User type is the level of membership of a user in a restaurant.*
Parameters:	*ReservationTypeRuleSet SASL, SecurityContext (injected by Jersey)*

Return Value:	*ReservationTypeRuleSet (echo input)* *NOTE: The ReservationTypeRuleset is an object that contains an array of TimeRange, MemberType, and ReservationType enums. This object is a domain level object and has no counterpart in the database. It will be marshaled into a domain string for storage.*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	*Chalkboards.setDomainVariable*
Data Access Objects:	*com.faralam.domains.restaurants.dao. AppointmentPolicy*
Business Logic:	*The entire ruleset object is marshaled into a string for storage in the database. In case of update, we simply replace the ruleset with the new one.*
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	*setReservationTypeAllowed (ReservationTypeRuleset, SASL, SecurityContext)*

queue length for pseudo-reservations

When pseudo-reservation is supported, the queue length is determined by computing the value from the QueueLengthRuleSet. The QueueLengthRuleSet will contain an array of TimeRange, , usertype, int (length) triplets. This object is marshaled into a string and unmarshalled for use in logic later.

Setting the Queue length in the form of a QueueLengthRuleSet object.

Use Case Details

Description:	Set the queue length for a given time range.
Actors:	User role must be 'Owner or SuperUser or Admin' in the security context.
Normal Flow:	The queue length is set.

setPseudoQueueLength

REST Mapping:	*[PUT][POST] /restaurant/queuelength*
Description:	*The queuelength computation rules are provided as an object, the QueueLengthRuleSet, which contains all the information necessary to compute the current queue length for the user at this time.*
Parameters:	*QuelenthRuleSet, SASL, SecurityContext*
Return Value:	*QuelenthRuleSet*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	*Chalkboards.setDomainVariable*
Data Access Objects:	*com.faralam.domains.restaurants.dao. AppointmentPolicy*
Business Logic:	
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	*setQueueLength(QueueLengthRuleSet,SecurityContext)*

Retrieving the QueueLength for current user at current time

Use Case Details

Description:	Retrieves the queue length given a time or time range
Actors:	User role must be 'Owner or SuperUser or Admin' in the security context.
Normal Flow:	The queue length is returned. UnableToComplyException if queue length is not defined.

API Details

REST Mapping:	*[GET] /restaurant/queuelength*
Description:	*NOTE: The time or time range is provided as a string. A single time is treated as a point in time, while a pair of times is treated as a time range.*
Parameters:	*SASL, TimeRange, (and SecurityContext injected by Jersey)*
Return Value:	*QueueLength*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	*Chalkboards.setDomainVariable*
Data Access Objects:	*com.faralam.domains.restaurants.dao.AppointmentPolicy*
Business Logic:	*The API retrieves the SeatingPolicy and determines if the current time or time range falls in the given policy, and then determines the queue length for that timerange.for that usertype. Note that the object exists only in the domain layer, there is no counterpart in the Chalkboards layer.*
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	*getQueueLength(UserType, SASL, TimeRange,SecurityContext)*

Delete QueueLength

Given an SASL, this method simply deletes the domain variable entry.

RESERVATION

We have two basic type of Appointments, Pseudo-Appointments and Appointments (real appointments). These are again divided into different types based on the details of the appointment relations.

In many cases, the available ‘seats’ or available timeslots are never actually generated, but rather they are calculated on demand based on the TimeSlotPolicy or the SeatingPolicy.

For example, an SASL has to decide on a queue length, which is updated via the TimeSlotPolicy of an SASL. This can be updated dynamically. A pseudo-reservation is made only if the timeslot policy indicates a positive number which is higher than the current number of pseudo-reservations in the appointment tables.

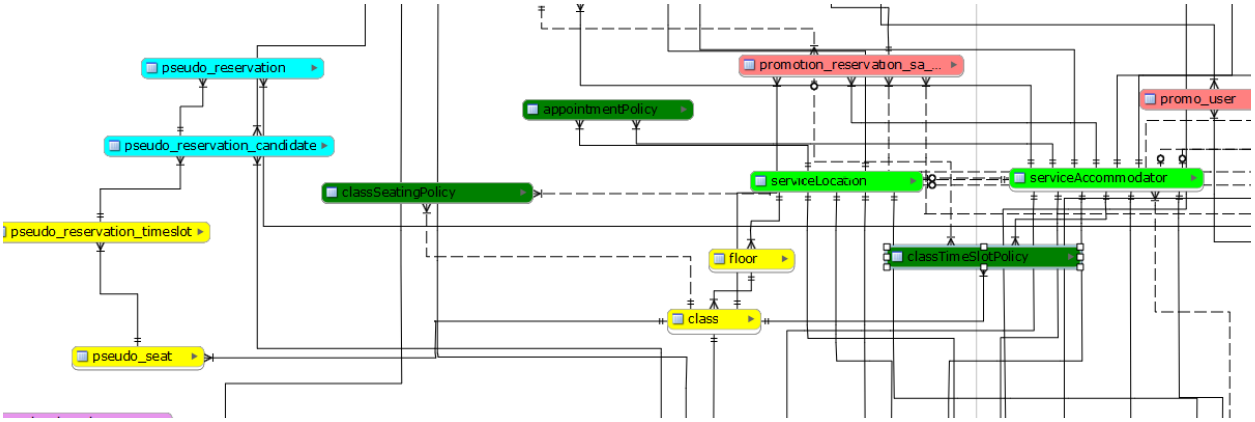
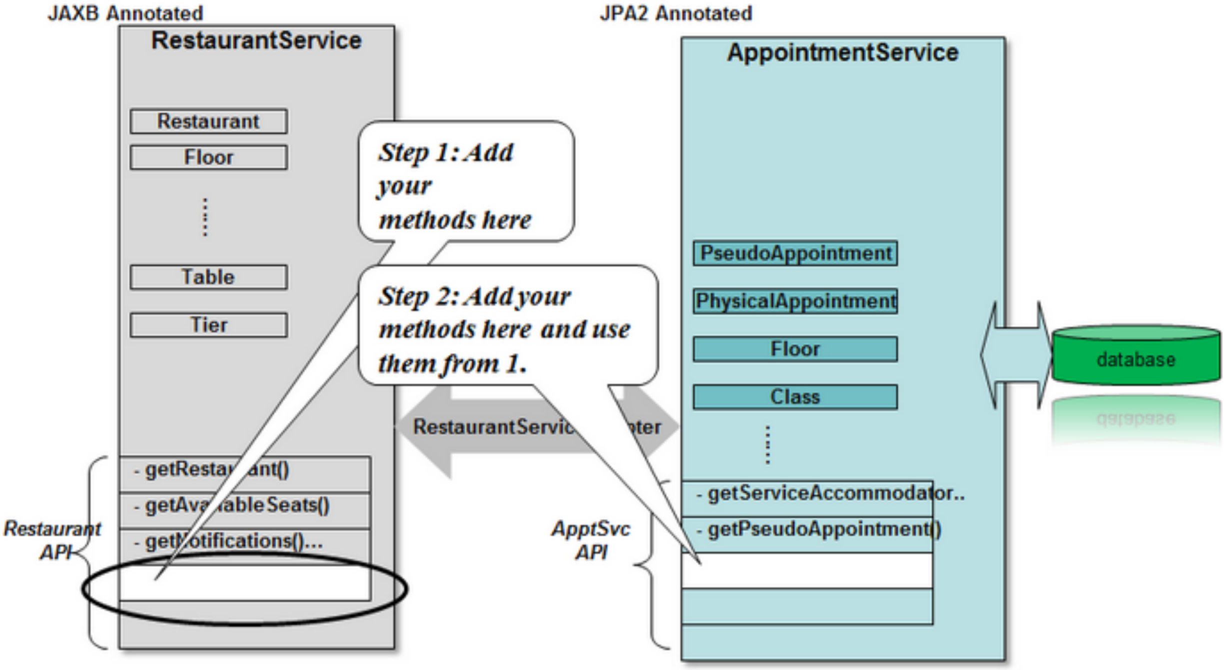


Figure 15 Excerpt from the schema showing the tables involved in Pseudo-Reservation. Note 'ClassTimeSlotPolicy'.



Step 1: Add your methods here

Step 2: Add your methods here and use them from 1.

Figure 16 Note that RestauratService is marshaled, so the objects are JAXB annotated. Chalkboards classes are JPA2 annotated, for database synchronization.

Common methods

getReservationsForUser

REST Mapping:	*[GET]/restaurant/ getReservationsForUser*
Description:	*For the given USER_ID, returns a list of Reservations, possibly empty*
Parameters:	*USER_ID*

Return Value:	*An Array of Reservation Objects.*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	*getReservationsForUser (USER_ID, SecurityContext)*

getReservations

Description:	retrieves list of reservations for a restaurant
Actors:	User role must be 'Owner or SuperUser or Admin' in the security context.
Normal Flow:	List of reservations is returned.

getReservations

REST Mapping:	*[GET] /restaurant/* getreservations
Description:	
Parameters:	*SA,SL, startTime,endTime,USER_ID, token*
Return Value:	*List of Reservations*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	*We first determine the queuelength, then determine how many appointments have already been made by querying the PseudoAppointment table, the return the difference r zero if negative.*
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	getReservations *(SA,SL, , startTime,endTime USER_ID,token,.SecurityContext)*

Pseudo-reservation API

A pseudo reservation or PseudoAppointment can be in one of many states, -as determined by a String (a domain variable). The string is arbitrary to Database.

Transient, Pending, Approved, Rejected, Consumed.

Transient: When the user is still deciding whether to make the reservation or not.

Pending: when the user has proposed, but SeatingPolicy says that appointments need approval.

Approved:

Rejected

Consumed: System did modify the state as consumed.

Retrieves the current pseudo reservations for the given SASL, time-range.

AvailableTableCount object.

This object encapsulates all the information necessary for an App to

getAvailablePseudoReservationsCount

Description:	Retrieves the current available pseudoreservation count left.
Actors:	User role must be 'Owner or SuperUser or Admin' in the security context.
Normal Flow:	AvailableTableCount object is returned.

API Details

REST Mapping:	*[GET] /restaurant/* pseudoreservationcount
Description:	
Parameters:	*SASL, level, startTime,endTime,USER_ID, token*
Return Value:	*QueueLength*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	*com.faralam.domains.restaurants.dao.AppointmentPolicy, PseudoAppointment etc.*
Business Logic:	*We first determine the queuelength, then determine how many appointments have already been made by querying the PseudoAppointment table, the return the difference r zero if negative.*
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	getAvailablePseudoReservationsCount *(SA,SL, level,USER_ID,token, startTime,endTime,.SecurityContext)*

Types needed: A PseudoReservation Object must be created on the domain layer. This object will encapsulate the entire reservation context (userid, timerange etc). The PseudoReservation object will contain the state of the reservation.

NOTE: we do not create PseudoSeats in advance. We simply add a new seat if no seat is available.

addPseudoReservation

REST Mapping:	*[POST] /restaurant/* pseudoreservation
Description:	
Parameters:	*SASL, usertype, timerange NOTE: timerange is in the form or a PrioritizedTimeRange string.*
Return Value:	*PseudoReservation(echo)*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	*Variouse apis affecting PseudoSeat, PseudoTimeSlot, PseudoAppointment etc.*
Data Access Objects:	*com.faralam.domains.restaurants.dao.AppointmentPolicy, PseudoAppointment etc.*
Business Logic:	*We first determine the queuelength, then determine how many appointments have already been made by querying the PseudoAppointment table, the return the difference r zero if negative.*
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	addPseudoReservation *(SASL,PseudoReservation,SecurityContext)*

Modify the pseudo reservation. For example we can modify the timeRange (need verification?) state of the pseudoreservation when the user consumes the reservation, or cancels it etc. A reservation is interpreted as expired in the logic layer if the reservation was never consumed and the current time is passed the timerange. We do not have a separate state for expired reservations.

updatePseudoReservation

REST Mapping:	*[PUT] /restaurant/* pseudoreservation
Description:	
Parameters:	*SASL, PseudoReservation (modified)*
Return Value:	*PseudoReservation (echo)*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	*Variouse apis affecting PseudoSeat, PseudoTimeSlot, PseudoAppointment etc.*
Data Access Objects:	*com.faralam.domains.restaurants.dao.AppointmentPolicy, PseudoAppointment etc.*
Business Logic:	
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	updatePseudoReservation *(SASL,PseudoReservation,SecurityContext)*

deletePseudoReservation

REST Mapping:	*[DELETE] /restaurant/* pseudoreservation
Description:	
Parameters:	*SASL, PseudoReservation (modified)*
Return Value:	*PseudoReservation (echo)*

Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	*Variouse apis affecting PseudoSeat, PseudoTimeSlot, PseudoAppointment etc.*
Data Access Objects:	*com.faralam.domains.restaurants.dao.AppointmentPolicy, PseudoAppointment etc.*
Business Logic:	
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	updatePseudoReservation *(SASL,PseudoReservation,SecurityContext)*

Retrieve All pending PseudoReservations

REST Mapping:	*[GET] /restaurant/* pseudoreservations
Description:	*Retrieves all entries from the PseudoAppointmentTable, converts them to PseudoReservations, and returns the array.*
Parameters:	*SASL, TimeRange (modified)*
Return Value:	*Array of PseudoReservations*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	*Variouse apis affecting PseudoSeat, PseudoTimeSlot, PseudoAppointment etc.*
Data Access Objects:	*com.faralam.domains.restaurants.dao. PseudoAppointment etc.*
Business Logic:	
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	getPseudoReservations *(SASL,timeRange,SecurityContext)*

PROMOTIONS SERVICE (API DETAILS)

A **'Promotion'** is a picture along with necessary annotations that can be shown to the phone/web client and who can book it (if bookable) or get it scanned (if it is scannable) etc. Promotions are associated with SASL (ServiceAccommodator, ServiceLocation) and Members using other tables. Promotions are time-sensitive as well as user/location sensitive. In other words, Promotions have restrictions.

The Promotion Service allows us to create, modify, delete and track usage of Promotions. The following sections describe a Promotion object in detail, followed by the API methods needed to access the Promotion service.

Promotion Object

A Promotion object has the following variables.

Relational Variables : PromotionID (primaryKey) (this is determined by the server during creation.

Non-Relational Variables: (These are all mapped to appropriate Domain Strings. Multiple Variables can be combined into single Domain Strings if necessary, since we have only 10 Domain Strings available).

1. PromotionCode (String): This is code that is provided by the Client. It could be generated by our own client, or it could be taken from some other external system.

2. DisplayText (String) This is the Promotion Banner text. For example, “Half of Deserts!”

3. DisplayTextLocation(String) : The x,y coordinates, relative percents, that determine where on the promotion the display text will be located (top-left corner of text).

4. Action (String): If any action is available for the Promotion. Usually it is ‘bookable’ or ‘scannable’.

5. ActionText (String): The text. For example, if it is ‘bookable’, this will be printed on the bookable button.

6. Restrictions (String): A comma separated list of restrictions for this promotion. For example, ‘members-only’, or ‘weekends’, or ‘zone-1’ or ‘off-peak’, ‘multi-use’ etc. (NOTE: By default all promotions are single-use unless otherwise specified in ‘restrictions’)

7. MainPicture (byte[]) this is the main picture in png format, of the Promotion. This picture will be maximum **200x300 pixels.**

8. OverlayPicture (byte[]) this is optional. If present, this is usually a scannable bar code. This could be added to the promotion by the server at runtime also.

9. OverlayPictureLocation(String): The x,y coordinates, relative percents, that determine where on the promotion the display text will be located (top-left corner of text).

10. PrioritizedTimeRange:(String) (this is the string representation of a PrioritizedTimeRange object that we use throughout the system. The reason for this specific format is to standardize time range notations. It consists of the following in specific formats (for testing any format can be used)

11. StartTime (String) This is a formatted Date-Time String that indicates the beginning of validity of the Promotion

12. EndTime(String) This is a formatted Date-Time String that indicates the beginning of validity of the Promotion.

DataBase Tables

The following database tables required for Promotions (included the *membership* tables for information only)

1. promotion : Promotions are entries in the Promotions table. Each promotion is identified by {PromotionID } as the primary key. the promotion table contain the Promotion id and all the other details stored in domain variables

2. promotion_spam : Promotion spam table contains SA SL ids , user can see these promotions if he clicks on number of notifications next to Restaurant name on Restaurant list page

3. promotion_*membership* : This table contains promotions for specified members only

4. member : member table contains Userid, password , email , zip code and phone numbers of members

5. *membership* : *membership* table contains SA SL Id, role (restaurant owner, chef, dinner and moderator)

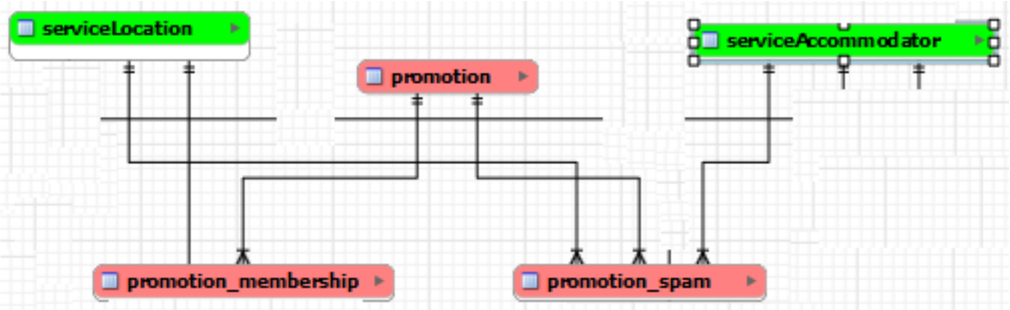


Figure 17 Excerpt from Schema showing basic relationship between Promotion and other relations (the relation lines are not correct, ... they got cut off. Please see main schema for details.)

Recall that an SA,SL key pair identifies a business uniquely in our system. This foreign key pair is the only connection of the Promotions feature to the rest of the Chalkboards APIs.

Data Access Objects (DAO)s

The DAO's required for above tables are in the Promotions domain.

- Promotion.java
- PromotionSpam.java
- PromotionMembership.java
- Member.java
- Membership.java

Promotions API

Add a Promotion

Promotions are added by the business owner (e.g. Restaurant Owner)

A promotion is created in the 'proposed' state. At a later time, the promotion moves to the Approved state. Until then, the promotion will not be displayed in any UI. However, the promotion can still be retrieved, modified, etc.

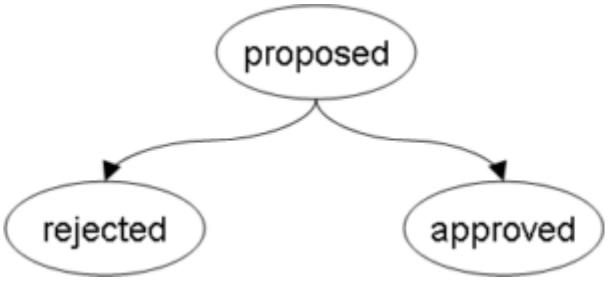


Figure 18 Basic state model for the creation/apporval of a Promotion. On creation, a promotion is in the

Proposed state. Upon review by our staff, the Promotion is either Approved or Rejected.

promotion

Description:	A promotion is created by the client (via a Portal or other interface) and added to the system.
Actors:	User role must be 'Owner' in the security context.
Normal Flow:	The promotion is sent to the server which, upon successful addition, returns the promotion object back. The returned promotion object includes the promotion ID as well as any modifications made to the artwork.

REST Mapping:	*[POST] /promotions/promotion*
Description:	*a PromotionCandidate object is received by this method.*
Parameters:	*PromotionCandidate, (and SecurityContext injected by Jersey)*
Return Value:	*Promotion*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or* *If the promotion contains invalid parameters, or invalid dates or invalid artwork format.* *PanicException is thrown if there is an internal server error.*
Member of:	*com.faralam.domains.promotions.api.PromotionService*
Syntax:	*addPromotion(PromotionCandidate,SecurityContext)*
Core API Counterpart:	*addPromotion*
Data Access Objects:	*com.faralam.domains.promotions.dao.PromotionDAO*
Business Logic:	

Update a Promotion

Usually invoked to modify an existing promotion.

Description:	A promotion is updated either by the client of by the service.
Actors:	User role must be 'Owner or SuperUser or Admin' in the security context.
Normal Flow:	The promotion is sent to the server which, upon successful update, returns the promotion object back.

promotion

REST Mapping:	*[PUT] /promotions/promotion*
Description:	*a Promotion object is received by this method.*
Parameters:	*Promotion, (and SecurityContext injected by Jersey)*
Return Value:	*Promotion*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or* *If the promotion contains invalid parameters, or invalid dates or invalid artwork format, or if the Promotion does not exist in the DataBase.* *PanicException is thrown if there is an internal server error.*

Core API Counterpart:	*updatePromotion*
Data Access Objects:	*com.faralam.domains.promotions.dao.PromotionDAO*
Business Logic:	
Member of:	*com.faralam.domains.promotions.api.PromotionService*
Syntax:	*updatePromotion(Promotion,SecurityContext)*

Retrieve all promotions for an SASL

- getPromotions(SA, SL, Status=’Approved’)
- getPromotionsByUID
- getPromotionsByPID

promotions

Description:	User wants to see ALL promotions for a particular establishment
Actors:	User role can be anything.
Normal Flow:	The system returns a possibly empty list of promotions.

getPromotions

REST Mapping:	*[GET] /promotions/*
Description:	*This method returns all promotions that match the status (‘Approved’ by default)*
Parameters:	*SA, SL, startTime,endTime,USER_ID,level,status,Token*
Return Value:	*List of Promotions*
Exceptions:	*UnableToComplyException is thrown invalid SA, SL or status are supplied* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	*com.faralam.domains.promotions.dao.PromotionDAO*
Business Logic:	
Member of:	*com.faralam.domains.promotions.api.PromotionService*
Syntax:	*getPromotions(SA, SL, Status=”approved’)*

Retrieve all promotions for USER_ID

getPromotionsByUserLocation

Description:	User wants to see promotions that are specific for him for his location
Actors:	The user must be a valid member of the restaurant or establishment.
Normal Flow:	The system returns a possibly empty list of promotions.

API Details

REST Mapping:	*[GET] /promotions/getPromotionsByUserLocation?UID=’xx’&location=’yy’*
Description:	*This method returns all promotions that match the USER_ID.*

Parameters:	*USER_ID, location*
Return Value:	*List of Promotions*
Exceptions:	*UnableToComplyException is thrown invalid SA, SL or status are supplied, or if the User is not a member at all.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	*com.faralam.domains.promotions.dao.PromotionDAO*
Business Logic:	*The idea is to allow the restaurant to target specific members for Promotion. A USER_ID is translated to membership ID if available.* *All other rules/restrictions for promotions are the same.*
Member of:	*com.faralam.domains.promotions.api.PromotionService*
Syntax:	*getPromotionsByUserLocation (USER_ID, Location,SecurityContext)*

getPromotionsForTimeRange

Description:	User wants to see promotions that are specific SA,SL and timeRange
Actors:	The user must be a valid member of the restaurant or establishment.
Normal Flow:	The system returns a possibly empty list of promotions.

API Details

REST Mapping:	*[GET] /promotions/getPromotionsForTimeRange?...*
Description:	*This method returns all promotions that match the SA,SL and other parameters*
Parameters:	*SA,SL,startTime,endTime,level,status,USER_ID,Token*
Return Value:	*List of Promotions*
Exceptions:	*UnableToComplyException is thrown invalid SA, SL or status are supplied, or if the User is not a member at all.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	*com.faralam.domains.promotions.dao.PromotionDAO*
Business Logic:	*For a restaurant to check what promotions it is running.*
Member of:	*com.faralam.domains.promotions.api.PromotionService*
Syntax:	*getPromotionsForTimeRange (USER_ID, Location,SecurityContext)*

Description:	User wants to see promotions that are specific for him (user=USER_ID) for a particular establishment.
Actors:	The user must be a valid member of the restaurant or establishment.
Normal Flow:	The system returns a possibly empty list of promotions.

getPromotionsByUID

REST Mapping:	*[GET] /promotions/promotionByUID?UID='xx'&SA='xx'&SL='xx'&status='xx'*
Description:	*This method returns all promotions that match the USER_ID. Note that if the USER_ID is not a member then an exception will be thrown.*
Parameters:	*PID, SA, SL*
Return Value:	*List of Promotions*
Exceptions:	*UnableToComplyException is thrown invalid SA, SL or status are supplied, or if the User is not a member at all.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Member of:	*com.faralam.domains.promotions.api.PromotionService*
Syntax:	*getPromotionsByUID(SA, SL, USER_ID, Status="approved')*
Data Access Objects:	*com.faralam.domains.promotions.dao.PromotionDAO*
Business Logic:	*The idea is to allow the restaurant to target specific members for Promotion. A USER_ID is translated to membership ID if available.* *All other rules/restrictions for promotions are the same.*

Retrieve a specific Promotion

Use Case Details

Description:	User knows a PID and wants to retrieve the Promotion
Actors:	The user can be anyone.
Normal Flow:	The system returns the promotion or throws an exception

API Details

REST Mapping:	*[GET] /promotions/promotionByPID?PID='xxx"*
Description:	*This method returns all promotions that match the USER_ID. Note that if the USER_ID is not a member then an exception will be thrown.*
Parameters:	*PID*
Return Value:	*The Promotion*
Exceptions:	*UnableToComplyException is thrown if invalid PID is supplied* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	*com.faralam.domains.promotions.dao.PromotionDAO*
Business Logic:	
Member of:	*com.faralam.domains.promotions.api.PromotionService*
Syntax:	*getPromotionByPID(PID')*

Mark Promotion as used by member

We would like to know when a user has used up a promotion. To do this we use the promotion_*membership* table. This table is used both for member-specific promotions (new or used).

Use Case Details

Description:	User knows a PID and wants to retrieve the Promotion
Actors:	The user can be anyone.
Normal Flow:	The system returns the promotion or throws an exception

API Details

REST Mapping:	*[PUT] /promotions/ membershipID=’xxx”&UID=’xx’&state=’xx’*
Description:	*This method marks a Promotion as used by a member. If the promotion was ‘spam’ promotion (meant for all users) then a new entry is made in the table. Otherwise, the original entry, for membership specific promotions, is modified to indicate that the promotion is used. When a user makes a booking, we set the state to INPROCESS, later, we set the state to USED, when the restaurant or business acknowledges that the reservation/booking was consumed.*
Parameters:	*PID, USER_ID, State (NOTE: the state can be VALID, INPROCESS, USED)*
Return Value:	*The Promotion*
Exceptions:	*UnableToComplyException is thrown if invalid PID is supplied* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	*com.faralam.domains.promotions.dao.PromotionDAO*
Business Logic:	
Member of:	*com.faralam.domains.promotions.api.PromotionService*
Syntax:	*updatePromotionMembership(PID, USER_ID, State=’USED’)*

MEDIA SERVICE (API DETAILS)

This service provides all the media (gallery, slides, audio, video) related APIs.

getAllSlidesForSASL

REST Mapping:	*[GET] /media/getSlidesForSASL?serviceAccommodatorId=1&serviceLocationId=1*
Description:	*All slides for the SA, SL*
Parameters:	*SA, SL*
Return Value:	*List of Slides*
Exceptions:	*UnableToComplyException is thrown invalid SA, SL or status are supplied* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Member of:	*com.faralam.domains.media.api.MediaService*
Syntax:	*getAllSlidesForSASL (SA, SL)*
Data Access Objects:	

Business Logic:	

getSlideForSASL

REST Mapping:	*[GET]media/getSlideForSASL?serviceAccommodatorId=1&serviceLocationId=1&currrentIndex=1*
Description:	*Slides for the SA, SL*
Parameters:	*SA, SL, index (goes from zero to max)*
Return Value:	*Slide*
Exceptions:	*UnableToComplyException is thrown invalid SA, SL or status are supplied* *PanicException is thrown if there is an internal server error.* *NoSuchObjectFoundException is this is the last slide.*
Core API Counterpart:	
Member of:	*com.faralam.domains.media.api.MediaService*
Syntax:	*getAllSlidesForSASL (SA, SL)*
Data Access Objects:	
Business Logic:	

LIVEUPDATE SERVICE (API DETAILS)

We separate the status update related into a different domain. There are no specific tables for this in the database. The possible status options for an SASL are simply stored in a domain variable in the SASL. However, we expose this API as a separate service because this API is functionally important and distinct for us.

getStatusOptions

REST Mapping:	*[GET] /liveupdate/* statusoptions
Description:	
Parameters:	*SA,SL, startTime,endTime,USER_ID, token*
Return Value:	*list of StatusOption objects*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*

Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	getStatusOptions *(SA,SL,USER_ID,token,.SecurityContext)*

getStatus

REST Mapping:	*[GET] /liveupdate/* status
Description:	
Parameters:	*SA,SL, startTime,endTime,USER_ID, token*
Return Value:	*Status*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	getStatuss *(SA,SL,USER_ID,token,.SecurityContext)*

updateStatus

REST Mapping:	*[PUT] /liveupdate/* status
Description:	
Parameters:	*SA,SL,Status startTime,endTime,USER_ID, token*
Return Value:	*Status*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.domains.restaurant.api.RestaurantService*
Syntax:	getStatuss *(SA,SL,Status,USER_ID,token,.SecurityContext)*

COMMUNICATION SERVICES

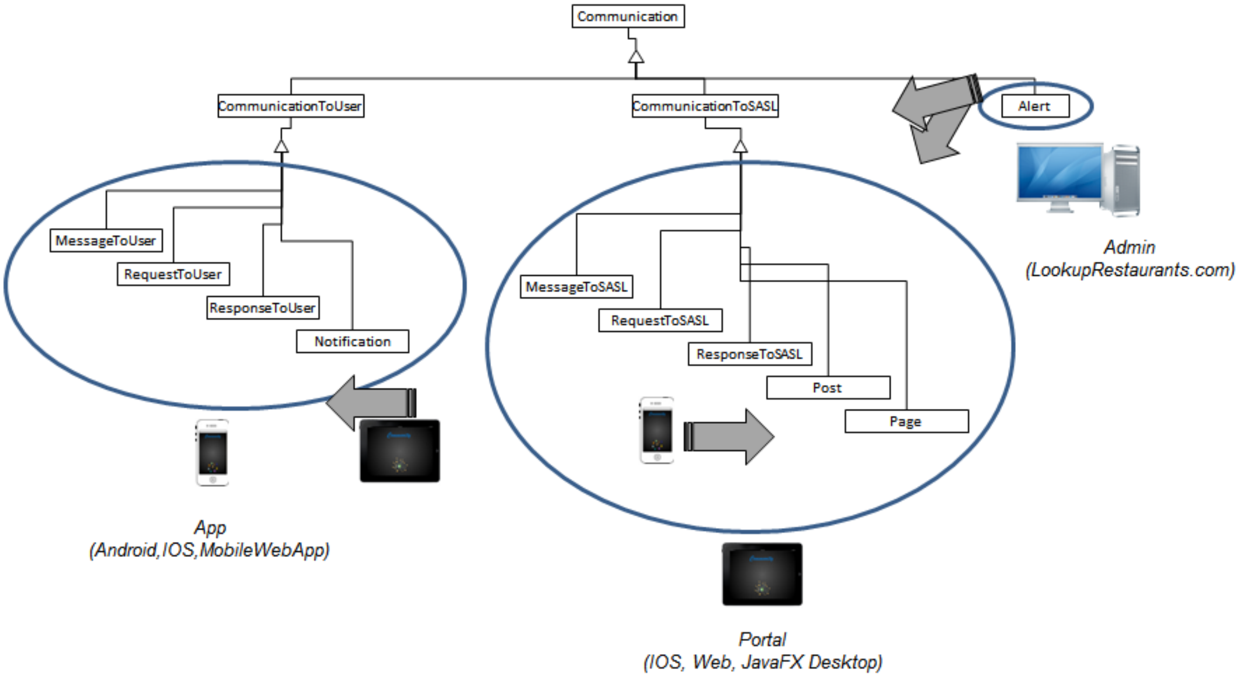


Figure 19 Communication Hierarchy.

The CommunicationServices encapsulate all communications between all users and restaurants etc.

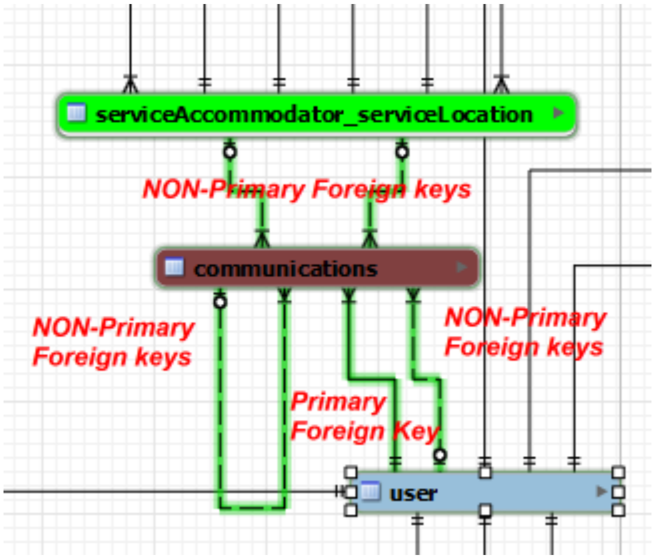


Figure 20 Schema excerpt for Communication entity. Note that there are lots of non-foreign keys. Only the

CommunictionID and the UserID of the author (AuthorID) are primary keys and thus required. The presence of the rest of the keys are enforced by the API (logic layers)

A Communication object must have the author and the Communication_ID specified and the combination must be using. The rest of the foreign keys are used to differentiate between types of messages. Note that there is an enumerated text field , CommunicationType that must be one of (NOTIFICATION, POST, MESSAGE, ALERT, PAGE etc.).

Communications can be of the following types:

1) Message: This is the simplest form of communication. It MUST have a toUserId (or the recipient USER id) OR it must have the toSA/toSL specified. The rest of the foreign keys must be null. When a restaurant is having a conversation with a person (member) then the toSASL/fromSASL will be filled out as appropriate, and this consistency must be checked by the API.

2) Posts: Posts are meant for a Restaurant's Wall or Blog.

3) Alerts: Alerts are meant for alerting users or restaurants about some special situation or service outage etc.

4) Pages: A restaurant owner may allow certain VIP members to be able to Page the restaurant for attention while he is inside the restaurant.

5) Notifications: Notifications are used by Restaurants to let the customer user know about some change of state. For example, table-ready notification, table-delay notification, traffic jams, parking issues, etc. etc.

We differentiate between all these types entirely at the API level. Internally, for the service layer, they are all 'Communication' objects.

getPostsForMember

REST Mapping:	*[GET]/ communication/getMessagesForMember*
Description:	*For the given USER_ID, returns a list of messages*
Parameters:	*USER_ID*
Return Value:	*An Array of Post Objects.*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	*tbd*
Business Logic:	
Member of:	*com.faralam.domains. communication.api.CommunicationService*
Syntax:	*getPostsForMember (USER_ID, SecurityContext)*

getPosts

REST Mapping:	*[GET]/ communication /getPosts*
Description:	*For the given USER_ID, returns a list of messages*
Parameters:	*SA,SL, startTime,endTime,USER_ID,Token*
Return Value:	*An Array of Post Objects.*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	*tbd*
Business Logic:	
Member of:	*com.faralam.domains. communication.api.CommunicationServices*
Syntax:	*getPosts (SA,SL, USER_ID,Token, SecurityContext)*

getNotificationForUser

REST Mapping:	*[GET]/ communication / getNotificationForUser*
Description:	*For the given Location and USER_ID, returns a list of notifications*
Parameters:	*Location, USER_ID*
Return Value:	*An Array of Notification Objects.*
Exceptions:	*UnableToComplyException is thrown If the user is unauthorized or does not have the proper permission, or if the supplied arguments are invalid.* *PanicException is thrown if there is an internal server error.*
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	*com.faralam.domains. communication.api.CommunicationService*
Syntax:	*getNotificationsForUser (USER_ID, Location, SecurityContext)*

APPENDIX: PRIORITIZEDTIMERANGE AND TIMESLOTS

A prioritized time range is used throughout the Chalkboards APIs .Therefore, we pay special attention to the description of a prioritized time range object.

A prioritized time range object consists of an array of strings, each describing a time range. The Chalkboards parses these time range strings in a hierarchical manner. Each time range string describes either a specific time range or a time range rule. Specific time-ranges override rules .

For example, if we say,

Wed 7pm: this implies from Wed 7pm to infinity, -the Wed being the next Wed from current time.

Wed 7pm to Thur 8pm means this time range for every week to infinity.

Jan 3 7pm to Jun 4 8pm means a very specific range, and this will override all of the above.

The Time range api reads these strings from the database, parses them and builds a TimeRange object. Now, it can be queried to determine if a given time or time range falls within these rules.

[TBD] (copy details from other doc)

APPENDIX: TEMPLATES

Use case Template

Use Case ID:	Enter a unique numeric identifier for the Use Case. e.g. UC-1.2.1		
Use Case Name:	Enter a short name for the Use Case using an active verb phrase. e.g. Withdraw Cash		
Created By:		**Last Updated By:**	
Date Created:		**Last Revision Date:**	
Description:	[Provide a brief description of the reason for and outcome of this use case.]		
Actors:	[An actor is a person or other entity external to the software system being specified who interacts with the system and performs use cases to accomplish tasks. Different actors often correspond to different user classes, or roles, identified from the customer community that will use the product. Name the actor that will be initiating this use case (primary) and any other actors who will participate in completing the use case (secondary).]		
Trigger:	[Identify the event that initiates the use case. This could be an external business event or system event that causes the use case to begin, or it could be the first step in the normal flow.]		
Preconditions:	[List any activities that must take place, or any conditions that must be true, before the use case can be started. Number each pre-condition. e.g. 1. Customer has active deposit account with ATM privileges 2. Customer has an activated ATM card.]		
Postconditions:	[Describe the state of the system at the conclusion of the use case execution. Should include both *minimal guarantees* (what must happen even if the actor's goal is not achieved) and the *success guarantees* (what happens when the actor's goal is achieved. Number each post-condition. e.g. 1. Customer receives cash 2. Customer account balance is reduced by the amount of the withdrawal and transaction fees]		

Normal Flow:	[Provide a detailed description of the user actions and system responses that will take place during execution of the use case under **normal, expected** conditions. This dialog sequence will ultimately lead to accomplishing the goal stated in the use case name and description. 1. Customer inserts ATM card 2. Customer enters PIN 3. System prompts customer to enter language performance English or Spanish 4. System validates if customer is in the bank network 5. System prompts user to select transaction type 6. Customer selects Withdrawal From Checking 7. System prompts user to enter withdrawal amount 8. … 9. System ejects ATM card]
Alternative Flows: **[Alternative Flow 1 – Not in Network]**	[Document **legitimate** branches from the main flow to handle special conditions (also known as extensions). For each alternative flow reference the branching step number of the normal flow and the condition which must be true in order for this extension to be executed. e.g. Alternative flows in the *Withdraw Cash* transaction: 4a. In step 4 of the normal flow, if the customer is not in the bank network 1. System will prompt customer to accept network fee 2. Customer accepts 3. Use Case resumes on step 5 4b. In step 4 of the normal flow, if the customer is not in the bank network 1. System will prompt customer to accept network fee 2. Customer declines 3. Transaction is terminated 4. Use Case resumes on step 9 of normal flow Note: Insert a new row for each distinctive alternative flow.]

Exceptions:	[Describe any anticipated **error conditions** that could occur during execution of the use case, and define how the system is to respond to those conditions. e.g. Exceptions to the Withdraw Case transaction 2a. In step 2 of the normal flow, if the customer enters and invalid PIN 1. Transaction is disapproved 2. Message to customer to re-enter PIN 3. Customer enters correct PIN 4. Use Case resumes on step 3 of normal flow]
Includes:	[List any other use cases that are included ("called") by this use case. Common functionality that appears in multiple use cases can be split out into a separate use case that is included by the ones that need that common functionality. e.g. steps 1-4 in the normal flow would be required for all types of ATM transactions- a Use Case could be written for these steps and "included" in all ATM Use Cases.]
Frequency of Use:	[How often will this Use Case be executed. This information is primarily useful for designers. e.g. enter values such as 50 per hour, 200 per day, once a week, once a year, on demand etc.]
Special Requirements:	[Identify any additional requirements, such as nonfunctional requirements, for the use case that may need to be addressed during design or implementation. These may include performance requirements or other quality attributes.]
Assumptions:	[List any assumptions that were made in the analysis that led to accepting this use case into the product description and writing the use case description. e.g. For the *Withdraw Cash* Use Case, an assumption could be: The Bank Customer understands either English or Spanish language.]
Notes and Issues:	[List any additional comments about this use case or any remaining open issues or TBDs (To Be Determined) that must be resolved. e.g. 1. What is the maximum size of the PIN that a use can have?]

An API Template must include the following:

methodName:

REST Mapping:	*[REST Path mapping syntax]*
Description:	*[Detailed description of the API]*
Parameters:	*[JAXB mapping. Details of all the input parameters along with description of values that can be passed.]*
Return Value:	*JAXB Mapping. Details of the values returned by API after execution].*
Exceptions:	*[When to throw what. ALL APIs throw either Panic or UnableToComply]*
Core API Counterpart:	*[If any direct core api counter part is there. JPA2 Objects/Mapping]*
Member of:	*[Service name]*
Syntax:	*[Syntax (prototype) of API]*

Notes/Warnings:	*[\Special notes or warnings while using the API*
Business Logic:	*[Short Description of the Business logic and algorithms that this API will implement internally]*
Synopsis:	*[One line description of the API]*
Calling Sequence:	*[The order in which an API must be called]*
Usage Scenario:	*[Expected scenarios where the API can be used]*
Examples:	*[Most of the programmers makes best use of this place and are expert in copying the examples provided illustrating the use of APIs].*
Related APIs:	*[Information on the APIs that could be related to the API being documented]*
Data Access Objects:	*[daos, if any that are being marshalled/unmarshalled]*
Business Logic:	*[Short Description of the Business logic and algorithms that this API will implement internally]*

methodName

REST Mapping:	
Description:	
Parameters:	
Return Value:	
Exceptions:	
Core API Counterpart:	
Data Access Objects:	
Business Logic:	
Member of:	
Syntax:	

Change Management

Tracking risks and issues

[In the following table, track the risks and issues that you identified.]

Date recorded	Risk description	Probability	Impact	Mitigation plan

Change management process

Change management process steps

[Describe the process that your team will follow to document and approve changes to the project. If your team uses a change control document, identify how and when team members should fill it out.]

Change management process flow

[Create a flow diagram of your change process.]

Change control board (CCB)

[Identify who will serve on the CCB, which determines whether issues are within the current project scope and whether they should be addressed.]