Elko Tutorial

Draft Version 0.7
1-February-2016

What is Elko?

Elko is a suite of servers for hosting highly scalable, sessionful internet applications, along with a set of common platform technology upon which these servers are all based. Although the Elko server suite provides a complete solution for most server applications, the underlying platform technology can readily be used for the creation of additional types of servers should they be needed.

Elko is targeted at stateful applications, in contrast to traditional web servers, which are (nominally) stateless. Most real-world internet applications, and certainly those that attempt to deliver a real-time interactive experience, entail significant short term application state that must be maintained somewhere. Web servers attain significant scalability by virtue of their statelessness, which enables capacity expansion by simple server replication, but this scale comes at the expense of considerable added complexity and inefficiency once the need to maintain application state is taken into account. Moreover, much of this complexity is inflicted on application developers in ways that slow development and impede software maintainability. Elko servers, in contrast, maintain application state directly in an immediate and developer friendly way, and achieve massive scalability by other means.

The suite also includes a set of service frameworks and object classes that build on top of the basic server infrastructure to provide a platform for delivering real-time, geo-enabled multiplayer games to clients on mobile devices, as well as to clients in traditional web, PC, and console environments.

The Elko Server Suite

The Elko server suite consists of a family of different types of servers that cooperatively form a highly scalable application hosting system for real-time, stateful applications. The current server suite consists of seven different kinds of servers:

Context server — where applications run
Director — directs users to contexts and load balances the pool of context servers
Presence server — tracks user presence and manages social graphs
Gatekeeper — adds support for external user authentication and login handling
Repository server — provides persistent object storage services
Workshop — provides a place for arbitrary cross-server services to run
Broker — central admin and configuration management for all the other servers

The simplest possible configuration consists of a single *context server*. More context servers may be added for scale, at which point one or more *directors* are required. A single director can manage a very large number of context servers, but it is common to have at least two directors to provide redundancy for reliability and fault tolerance. The context servers and directors themselves can do basic user authentication, but more sophisticated authentication and user account schemes may require the addition of one more *gatekeepers*, which can integrate with external account management, identity, and authentication systems. In the same vein, *presence servers* enable integration with social graph data, both external (such as a user's Facebook friends) and internal (for example, a competitive game might choose to keep track of players' rivalries). Some services (such as transactional bank account management for virtual currencies) are a global to an entire game or application, but still make use of real-time state; *workshops* provide a place for these kinds of services to be hosted. If configured individually, all of these disparate servers would add up to a fair bit of administrative complexity. The *broker* provides a means of simplifying that, by providing a one-stop centralized administrative control point for managing a large cluster of servers. The broker enables other servers to be added and removed dynamically, and isolates the individual servers from concern with overall configuration details or with order-of-startup issues. It also provides a centralized facility for server monitoring and metrics collection.

The Context Server and The Elko Application Environment

The context server is the heart of the Elko server suite. It is the place where application-specific classes are loaded and run to realize the particular objects and behavior that distinguish one application from another. The remaining types of servers play important supporting roles in making the whole system work effectively, but, except for a few unusual situations, from an application developer's perspective they can mostly be ignored.

Elko applications are object-oriented, in a couple of different ways. The Elko object model is based on the *unum* abstraction. The unum model distinguishes between two senses of the concept of "object": objects in the object-oriented programming sense, consisting of bundles of software state and behavior that each exist in some computer's memory, and objects in the world model sense, consisting of discrete entities within some abstract space independent of any particular computational embodiment. We call the first kind of thing an "OOP object" or just an object. We call the second kind of thing an *unum* (Latin for "one thing"). We make this distinction because a distributed application, such as a virtual world or an online game, contains things such as player avatars, chat rooms, documents, trees, tools, weapons, buildings, animals — the list is endless, really — that have their own distinct identities in the scope of the application or game but independent of any particular computer. An unum may have concurrent manifestations, which we call *presences*, on each of the various distributed machines that make up the application's implementation (and these presences in turn themselves are typically implemented by OOP objects, or clusters of related OOP objects, underscoring the usefulness of the unum/OOP object distinction). For example, if you and I are interacting

with each other in a virtual world game, my avatar may exist in one form on my iPhone, in a slightly different form on your Android phone, and in a wildly different form on the server via which we jointly play the game. Nevertheless, my avatar is a single, designatable entity within the game world that has a unitary existence, even though its state and implementation is spread across all these platforms.

In its typical manifestation (such as in the Elko framework described here), an unum is realized by a *server presence* and some number of *client presences*. It is common for all the client presences to share the same underlying implementation, though there may be differences to accommodate disparate platforms (e.g. iPhone vs. Android, Mac vs. Windows). However, it is quite common also for the server and client presences to be extremely different from each other, often to the point of being coded in entirely different programming languages, as the roles of the client and the server are very different and thus so are the tools they require to get their respective jobs done. In particular, the server presence of an unum typically needs to communicate with numerous client presences, but each client presence needs only to communicate with the single server presence. The key abstraction for thinking about distributed computation this way is division of labor. (The division of labor idea also invites consideration of distributed architectures other than the strict client-server model, but that is far outside the scope of this document.)

Presences of an unum communicate by sending messages to each other. Note that this means that a particular message is addressed along two dimensions: one dimension is the identity of the unum that is the target of the message, and the other is the particular presence or presences of that unum to which the message should be delivered. This two-dimensional address notion is largely invisible to the client implementation, since the client really only knows about one other presence: the server. But the distinction becomes important when implementing a server presence, as there may be multiple client presences that it needs to interact with.

In Elko, all inter-presence messages are unidirectional and asynchronous. That is, sending a message is an immediate, non-blocking operation with no return value. This is in contrast to the many distributed programming frameworks based on variations of the remote procedure call (RPC) abstraction, such as Java RMI, Pyro, CORBA, XML-RPC, or, for that matter, HTTP, where a message is typically a request that blocks waiting for a reply. One of the fundamental rules of programming in the Elko server environment is that no operations are permitted that could block.

The Elko context server provides three fundamental classes of unums, from which all applications are built. These are the *context*, the *user*, and the *item*.

A *context* is a rendezvous and coordination point for communications among multiple parties. A context provides a scope for object addressing and message routing — a common frame of reference for interaction. Contexts are the basis of resource allocation on the server, hence the term “context server". The context abstraction is concerned with the things that are visible in common to a set of users. What a context represents in a

particular application depends, of course, on the application, but examples might include a chat room in a multi-user chat application, an auction in a real-time auction application, a poker table in a real-time gambling game, or a small neighborhood in a virtual world game.

A *user* represents an autonomous agent acting within a context. By "autonomous", we mean that its actions are controlled from outside the server. Usually this corresponds to a human being interacting with a client application. At any given time, a user is in some particular context. Multiple users may be in the same context together, enabling them to engage in multiparty interactions of various kinds. Associated with a user is a communications connection between the user's client and the context server. This connection carries the message traffic between the associated client's client unum presences and the server's server unum presences. Continuing the examples from the previous paragraph, users would represent the people chatting in the chat room, the bidders in the auction, the card players at the poker table, or the avatars walking around on the street in the neighborhood.

An *item* represents any other kind of object that embodies some aspect of the application. Depending on the application, there might be a lot of items, a few items, or no items at all. For example, the chat room mentioned above might not have any use for them. Items can represent very abstract things, such as conditional bids in an auction. Items can represent concrete things, such as playing cards or poker chips. An application can have very few kinds of items, or a very large number. For example, one might imagine the virtual world game being populated with an enormous variety of different possible objects, each of which would be a different kind of item.

Contexts, users, and items all exist in a containership hierarchy. A context can contain users and items, users can contain items, and items can (potentially) contain other items. The containership hierarchy is central to the mechanisms the context server uses for object persistence, and it also plays an important role in controlling the kinds of access that different users have to different objects.

Each unum has a unique identifier within the application, called a *ref* (short for "reference"), by which it may be named in messages. When a client sends a message over a connection to a server and the server receives it, the message is delivered to the server presence of the object named by the message as its target. Similarly, a server may send a message over a connection to a client, addressing the message to the client presence of some object. The server may direct a single message to some or all of the client presences of an unum; the server's message system will automatically take care of the message fan-out, using the various clients' respective connections.

As provided by the Elko framework out of the box, each of these three kinds of unums supports a basic message protocol that is concerned with its nature as one of these core abstractions. The classes that implement them on the server provide a rich but basic set of APIs concerned with things like message delivery, control over persistence, manipulation of the containership hierarchy, and so on, along with facilities for creating

new objects and destroying existing ones. However, this basic framework contains no application-specific functionality. Providing that is the responsibility of the application developer.

Instances of the core unum types acquire their application-specific personalities via application-specific objects known as *mods* (short, variously, for “modules" or “modifications", depending on your perspective). A mod is simply a bundle of behavior and state that is attached to some particular context, user, or item. The mod augments the unum to which it is attached with additional message protocols specific to whatever functionality it is that the mod provides. Some or all of a mod’s state may be persisted along with that of its associated unum. The Elko context server uses this sort of mix-in pattern, in contrast to subclassing, because our experience has shown that game and virtual world applications often require bundles of object functionality that are orthogonal to the otherwise natural class hierarchy that the objects might be organized into. (In addition, separating developer-provided state from framework-provided state by associating them via composition rather than via embedding makes it much easier for the object persistence mechanism to operate reliably in the face of application bugs.)

At any given time, a context is said to be either *active* or *inactive*. When it is inactive, it sits in a quiescent state in the persistent object store, which we call the *repository*. Typically, an inactive context is activated when a user attempts to enter it (by definition, in its inactive state a context contains no users), though other events on the server can, in principle, also activate a context if the application’s design demands it. Upon activation, the context is loaded from the repository into the context server’s memory, along with any items that the context contains (and any items *they* contain, and so on). Once active, messages sent to the context server addressed to the context or any of the users or items it contains will be delivered to their targets. The methods that handle these messages can do whatever they do according to the logic of their particular application, including modifying the states of the various objects, sending messages of their own to clients or to other objects on the server, and so on. When the last user leaves the context and any running processes that were accessing it have finished, the context is deactivated: any changes to the state of the context and it contents are written back out to the repository and the in-memory objects are released. In addition, the state of an active context or any or all of its contents may also be checkpointed to the repository at any time by deliberate action on the part of application code when it is important for critical state to be saved.

A similar story describes the activation cycle of a user unum. The only important difference is that the user state (and that of any items the user contains) is loaded from the repository when the user connects to the server and written back out again when the user leaves. Since in normal operation a user’s connection and disconnection is coupled directly with their entry to and exit from a context, a context’s activation cycle and a user’s activation cycle are closely associated. The key difference is that additional users may enter a context (and themselves be activated) after the context is already active, and the context may remain active after a user exits.

Another way to think of this is as an object-oriented analog to a traditional paged virtual memory system, especially as it would be experienced in non-file-oriented operating system such as Multics or KeyKOS: unum state is like a virtual memory page, the contents graph of a context or user unum is like a page frame, unum activation is like a page fault, and so on.

Contexts and items may also be created dynamically by the actions of application code in the server. Such objects may add to the persistent state of the application, or they may be entirely ephemeral, existing only so long as users are interacting with them and simply vanishing afterwards rather than being saved into the repository.

Developing an Elko application consists of defining the role or roles that contexts will play, defining the various kinds of items that will populate these contexts and what these items will do, and defining what affordances users will have for manipulating these items and each other. Central to this is specifying the message protocols that the various objects will participate in.

We have found that often the most effective approach for thinking about how an Elko application will work is to begin by considering the division of labor between client and server, manifested by the client-server protocols of the various unums involved. From these protocols, you can then work outward to the state and behavior that is needed on each side. On the server side, you develop mods for the contexts, users and items that give these objects their application-specific personalities. On the client side, you implement the appropriate user interfaces, graphical visualizations, and so forth to deliver the application or game experience to the human being who is actually interacting with the client device.

Nuts and Bolts

JSON Messaging

Communications between clients and the context server, as well as among the various different servers on the backend, is via *JSON messaging*.

JSON messaging is a set of conventions for encoding object-to-object messages in JSON. A JSON message is simply a JSON object of the form:

```
{ to:targetRef, op:verb, params... }
```

The property `to` designates the message target, i.e., the object to which the message is addressed. Its value is a *ref*, a string that uniquely names the target object in the scope of the messaging system on the arriving end of the communication.

The property `op` is the message verb, i.e., the operation code or method selector, a string that indicates to the message target which operation is to be performed upon receipt of the message.

Any number of other properties may also be included, and serve as named parameters. Their number, names, and meanings vary depending on the specific message being sent. They may be of any data type supported by JSON, as long as there is consistent mutual understanding between the sender and receiver of the proper content for the message. The order of the parameters is not significant, though in documentation we conventionally write the `to` and `op` parameters first for clarity of presentation. (Note that to improve documentation legibility, we also use a slightly augmented JSON syntax, where we skip the quotation marks around the property names. This notation is also understood by the server, as it considerably mitigates the drudgery of manual composition of JSON expressions during debugging. The server does, however, conform to the strict RFC 4627 JSON standard in the things it outputs unless explicitly configured not to.)

For example, the message:

```
{ to:"u-47-3699102", op:"say", utterance:"Hi Fred!" }
```

might be a chat message directed to another user.

JSON messages may be transmitted over any reasonable communications medium. The current implementation supports raw TCP, HTTP, WebSockets, and ØMQ, as well as the SSL variants of these (HTTPS, etc). We also support a protocol of our own that we've named Resumable TCP (RTCP), which adds a simple session layer for JSON messaging on top of TCP to support reliable communications over unreliable wireless networks, specifically for mobile devices.

A message connection is always a bidirectional, multiplexed, machine-to-machine (or process-to-process) message pipe. Two communicating machines, whether they are a client and a server or a server and another server, typically maintain a single connection between them, over which all message traffic on behalf of objects on either side is carried. A given connection is kept alive for the duration of the interaction between the two machines, until one side or the other decides to terminate it. This communications session is stateful, in that either party is permitted, indeed expected, to maintain continuity of state between successive messages. This is directly in contrast to stateless, sessionless protocols such as HTTP.

Though a connection is bidirectional, messaging over the connection is unidirectional and asynchronous. That is, as soon as a running application passes a message to the messaging system, it is able to continue on its own without waiting for reply or confirmation. When a message is part of a request-reply protocol, the reply must be treated as an explicit, separate return message by the two parties. Any such protocol is part of the design of the particular application in question and is not the business of the messaging system itself. Any given message may result in a reply from its recipient, no reply, or possibly even many replies, depending on the design of the application and its protocols.

Messages on a connection are ordered, in the sense that a sequence of messages transmitted over a given connection from machine A to machine B will be processed at machine B in the same order they were transmitted by machine A. However, this ordering is strictly on a per-connection basis. If there is more than one connection between two machines (not normal but not forbidden either), there are no guarantees about the relative order of messages sent over different connections.

Since HTTP is stateless and sessionless, an additional transport protocol is used when the message transport medium is HTTP. This protocol uses special URLs and additional JSON encoding within the HTTP request and reply bodies to efficiently synthesize a sessionful, symmetric, bidirectional, asynchronous message channel out of an ongoing series of transient, asymmetric, synchronous, RPC-style HTTP requests. The details of this transport protocol are documented elsewhere.

Object State Representation

For purposes of transmission in messages and writing to persistent storage, the state of an object is serialized into JSON following a few simple representational conventions. The conventional form of an object is:

```
{ type:typeTag, ref:refString, properties... }
```

The property `type` encodes the data type of the object. This is a simple tag string that is mapped to the actual object class internally. On the server, a mapping table, itself stored as a persistent object in the server's object repository, maps the type tag to the fully qualified Java class name of the class whose constructor will decode the JSON serialized representation into a live Java object in the server's memory. (We use a tag string rather than using the class name directly in order to decouple the JSON form from server-specific implementation details. It also reduces the size of the representation, cutting bandwidth and storage requirements.) Depending on circumstances, the `type` property is not always required, since the types of pure data objects embedded within other objects can frequently be determined by the context in which they are used. For example, say you had a polygon object, one of whose properties is an array of points, e.g.:

```
{ type:"poly", points:[point, point, ...], whatever... }
```

you might simply represent an individual point as:

```
{ x:xVal, y:yVal }
```

rather than, say:

```
{ type:"point", x:xVal, y:yVal }
```


since at the place in the code where it needs to deserialize the array of points, the code already knows that it's looking for an array of points, and redundantly tagging each of them would just be wasteful.

The property `ref` is the object's ref, the unique identifier that designates that specific object instance. The `ref` property is only present when the object in question is referenceable, i.e., if it can be pointed to by something else, most notably as the designated target of a message send. Pure data objects that are simply structured property values inside other objects are typically not referenceable in this way, and so their JSON representations would omit this property.

The remaining properties depend entirely on the object type. It is the responsibility of the code that serializes and deserializes a given type to interpret or generate these properties as appropriate.

Note that this JSON representation is a serialization scheme used for communicating an object's state from one place to another. What it becomes when interpreted by the receiving entity is entirely undefined here. We loosely talk about storing a serialized object in the repository, but how the repository actually represents things internally is its own business. It is not required, or even really expected, that the repository will be storing and retrieving actual literal JSON strings.

A Quick Tour of The Server Side of An Application

Let's illustrate the basic structure of an Elko context server application by working through a simple example: a minimal multi-user chat system. A chat server is Elko's version of the classic "hello world" application.

For this application, we'll implement a chat room as a context, with a simple context mod that realizes the chat functionality. Beyond the basic context functionality, all that is needed to have a working chat system is a message that allows a user to issue a chat utterance that is then echoed to all the other users in the room:

```
{ to:contextRef, op:"say", speech:textToBeSpoken }
```

which the mod then sends to everybody in the room as:

```
{ to:contextRef, op:"say", speech:textToBeSpoken,
  from:speakerRef }
```

Here is the complete server-side implementation of this application:

```
package com.example.tutorial;

import org.elkoserver.foundation.json.JSONMethod;
import org.elkoserver.foundation.json.MessageHandlerException;
```



```
import org.elkoserver.json.EncodeControl;
import org.elkoserver.json.JSONLiteral;
import org.elkoserver.json.Referenceable;
import org.elkoserver.server.context.ContextMod;
import org.elkoserver.server.context.Mod;
import org.elkoserver.server.context.User;

/**
 * A simple context mod to enable users in a context to chat with
 * each other.
 */
public class SimpleChat extends Mod implements ContextMod {
    @JSONMethod
    public SimpleChat() {
    }

    public JSONLiteral encode(EncodeControl control) {
        JSONLiteral result = new JSONLiteral("chat", control);
        result.finish();
        return result;
    }

    @JSONMethod({ "speech" })
    public void say(User from, String speech)
        throws MessageHandlerException
    {
        ensureSameContext(from);
        context().send(msgSay(context(), from, speech));
    }

    static JSONLiteral msgSay(Referenceable target,
                              Referenceable from,
                              String speech)
    {
        JSONLiteral msg = new JSONLiteral(target, "say");
        msg.addParameter("from", from);
        msg.addParameter("speech", speech);
        msg.finish();
        return msg;
    }
}
```

The declaration of the class:

```
public class SimpleChat extends Mod implements ContextMod {
```

subclasses `Mod`, which is the abstract base class for all mods. Among other things, this base class provides a bunch of support methods for common operations that mods tend to

want to do. The interface `ContextMod` is a marker interface that identifies this as a context mod, i.e., a mod that is intended to be attached to contexts. The server will require any mods that you attempt to attach to a context to implement this interface. There are analogous `UserMod` and `ItemMod` interfaces for users and items. Note that these are not mutually exclusive: something can be, say, both a user mod and item mod at the same time. Indeed there is a `GeneralMod` interface that declares the mod to be attachable to any kind of unum.

The constructor:

```
@JSONMethod
public SimpleChat() {
}
```

is invoked by the server's object deserialization mechanism when the JSON object descriptor for the mod is read from the repository as part of the descriptor for the context to which it is attached (more on this below). The `@JSONMethod` annotation marks it as being intended to be used in this way. This particular constructor is trivial because this mod has no state or parameterizable configuration; we'll present a slightly more complicated example below showing how to incorporate parameters and state.

The `encode()` method is another part of the serialization interface:

```
public JSONLiteral encode(EncodeControl control) {
    JSONLiteral result = new JSONLiteral("chat", control);
    result.finish();
    return result;
}
```

The signature for this method is defined by the `Encodable` interface, which the base class `Mod` is declared to implement but does not actually provide. Every mod class needs to provide an appropriate `encode()` method, whose job is to produce a suitable JSON serialization for instances of its class. Since the `SimpleChat` mod has no state, in this case serialization is very simple — all that is needed is the type tag property.

The `JSONLiteral` class is essentially a structured wrapper for a Java string buffer, in which a JSON literal string is incrementally constructed. Here, the `JSONLiteral` constructor call generates the preamble for a JSON object descriptor, and then the descriptor is immediately closed off by the call to the `finish()` method since nothing else is needed beyond the type tag. The resulting JSON string will output as:

```
{ "type":"chat" }
```

though you don't need to actually be aware of this.

Note the one parameter to the `encode()` method: an instance of `EncodeControl`. This indicates how the object is to be serialized. The most important serialization

consideration is usually whether the object is being sent to the client or to persistent storage. It is often the case that the server's persistent representation for an object includes information that is not to be shared with clients. Less commonly, there may be ephemeral state that is sent to clients but not actually stored. The `control` parameter lets the `encode()` method distinguish between these cases. In this example, there's no such distinction to be made and so the parameter is ignored. However, since the mod has no state, all that the mod descriptor conveys to the client is that the context supports chat. If the client were being written as a pure chat application, we might simply assume that any context it would be interacting with supports chat, and so we might prefer to save the bandwidth spent on telling the client something it already knows implicitly. In this case, we might rewrite the `encode()` method as:

```
public JSONLiteral encode(Encode control) {
    if (control.toClient()) {
        return null;
    } else {
        JSONLiteral result = new JSONLiteral("chat", control);
        result.finish();
        return result;
    }
}
```

Returning `null` from an `encode()` method says "there's nothing here to encode, ignore me". In this case the `SimpleChat` mod descriptor would be part of the persistent representation of the context, but the context descriptor that is sent to the client would not include it.

The `say()` method is where all the actual work of the mod gets done:

```
@JSONMethod({ "speech" })
public void say(User from, String speech)
    throws MessageHandlerException
{
    ensureSameContext(from);
    context().send(msgSay(context(), from, speech));
}
```

There are several things going on here that warrant explanation.

This is a *message handler method*, intended to be invoked by the context server upon receipt of a message from the client. In this case, it defines a handler for the `"say"` message, taking a single string parameter named `speech`. This matches the message protocol whose description we gave on page 9 at the beginning of this example.

The first parameter of any message handler method is always the user from whom the message was received. The remaining parameters, if any, are mapped from the position-independent named parameters found in the JSON message to the positional-but-

anonymous parameters of the Java method. The `@JSONMethod` annotation lists the names of the message parameters in order. This annotation is needed because although we humans can see the parameter names in the source code, the Java virtual machine (and the reflection API that goes with it) does not make parameter names available to executing code, hence this additional hint is required so that the message dispatch machinery can do its work.

Any message handler can fail with a `MessageHandlerException`, so all message handlers need to be declared to throw it.

The first line of the method:

```
ensureSameContext(from);
```

is a guard to ensure that the message didn't come from somebody outside the context. This prevents unauthorized users from injecting message traffic into the chat (other applications might have a use for messages originating from outside the context, so this guard is optional). The `ensureSameContext()` function is one of a number of guard functions provided by the `Mod` base class. It will throw a `MessageHandlerException` if the user given as its parameter is not present in the same context as the object to which the mod is attached.

The `context()` function is also provided by the `Mod` base class. It returns the `Context` object for the context in which the message is being handled. Passing a message to a `Context`'s `send()` method will send that message to all the users in the context. The `msgSay()` method here produces such a message:

```
static JSONLiteral msgSay(Referenceable target,
                          Referenceable from,
                          String speech)
{
    JSONLiteral msg = new JSONLiteral(target, "say");
    msg.addParameter("from", from);
    msg.addParameter("speech", speech);
    msg.finish();
    return msg;
}
```

It's worth mentioning that, in this example, a separate `msgSay()` method is not strictly needed, as the message is only sent from one place in the code and so could be generated inline. However, as a general rule it's a better practice, from the perspective of code clarity and modularity, to separate message generation operations into methods of their own even if the message in question is only sent from one place. And surprisingly often you end up later wanting to send the same message from elsewhere in the code, so it can pay off that way too.

Compare the `msgSay()` message generator method to the mod's `encode()` method. Both start out by initiating the construction of a `JSONLiteral` object. However, whereas the `JSONLiteral` constructor parameters in `encode()` were a string and an `EncodeControl` object, denoting, respectively the type tag and what audience the serialized representation is intended for, here the parameters are a `Referenceable` object and a string, denoting, respectively, the `to` and `op` parameters. `Referenceable` is an interface provided by the Elko runtime and implemented by all objects to which messages can be sent. These are exactly the objects that have refs, and it is that ref which gets serialized as the value of the `to` parameter.

Unlike the exemplar `encode()` method, the `JSONLiteral` produced by `msgSay()` has a couple of additional JSON properties. These are the parameters of the JSON message being generated, which are appended to the message with calls to the `addParameter()` method.

When the `say()` method calls `msgSay()`, it passes the context itself as the message target, the sender of the incoming `"say"` message as the `from` parameter, and the speech text extracted from the incoming message as the `speech` parameter.

Let's say the chat room is the context with the ref `"ctx-chatroom1"` and there are two users, Alice and Bob, designated `"u-alice"` and `"u-bob"` respectively. Alice's client might send to the server the message:

```
{ to:"ctx-chatroom1", op:"say", speech:"Hi everybody!" }
```

The context has the `SimpleChat` mod attached to it, so it handles the `"say"` message using the `say()` method shown above. This ends up generating the outbound message:

```
{ to:"ctx-chatroom1", op:"say", from:"u-alice",
  speech:"Hi everybody!" }
```

which the server sends out to the clients of everybody in the context, namely Alice and Bob.

Note that the outbound message is addressed to the context, same as the inbound message was. This is something that people new to the Elko architecture often find surprising at first, expecting it to be addressed to the user whose client it is being sent to. However, we consider the chat activity to be taking place in the context and have associated the message broadcast operation with the context itself rather than any particular user. Each client has its own internal model of what is going on in the context, and sees this message as an action in that context also — conceivably a client could be participating in multiple chats in different chat rooms on the same server, for example, and the message address disambiguates where a particular utterance was made.

One obvious question you may be asking at this point is: where did the context come from in the first place?

The context is described by a JSON object descriptor in the repository. In the case of our example, it would be something like this:

```
{
    type:"context",
    ref:"ctx-chatroom1",
    name:"Example Chat Room",
    capacity:20,
    mods: [
        { type:"chat" }
    ]
}
```

This defines a context referred to as `"ctx-chatroom1"`, with our simple chat mod attached. In addition to the collection of mods, the `Context` class also supports a number of built-in application-independent properties that can be possessed by any context. In this case we've given it a human readable label and a limit to the number users allowed in at once.

The context server understands certain messages directly. There is a built-in pseudo-object named `"session"` that understands messages that manage a client's connection session. In particular, the `"entercontext"` message is used to enter a context.

Let's say we have a context defined as above, and further that user Bob is already there. Here's what the message dialog between Alice's client and the server might look like. We'll indicate messages from Alice's client to the server with "A→" and messages from the server to Alice's client with "A←" (and we'll similarly denote messages between the server and Bob's client with "B→" and "B←", and messages from the server to both of them with "AB←").

Alice first tells the server that she wants to enter the chat room context:

```
A→ { to:"session", op:"entercontext", context:"ctx-chatroom1",
   user:"u-alice"}
```

Note that the `"entercontext"` message is sent to `"session"` rather than to the context itself. This is for a couple of reasons. One is that the context might not actually be active on the server when she does this, and so there would be no context object there to receive the message in that case, but the `"session"` pseudo-object is always there, by definition. Also, it is possible that the server might choose to put her in a different context, especially if the context is very busy and the context cloning feature is enabled.

Also, this example doesn't involve of any kind of user authentication. The client claims to be whoever it chooses, and the server just accepts this. A real application would most likely want to require some kind of authentication before letting somebody in. The

`Session` and `Context` classes provide support for this, but that's outside the scope of this example.

In our example here, the server is happy to let Alice in, and so it puts her into the context that was requested. The first part of this is to inform her client about the current state of the context, by sending a series of `"make"` messages, directing the client to create its own local presences of the various objects that make up the chat room and its contents. The convention is that a `"make"` message is sent to the container into which the object it describes should be placed. The `"make"` message for the context itself is addressed to the session:

```
A← { to:"session", op:"make",
     obj:{ type:"context", ref:"ctx-chatroom1",
           name:"Example Chat Room", mods:[{ type:"chat" }] }}
```

And then another `"make"` message creates a presence of Bob inside the context:

```
A← { to:"ctx-chatroom1", op:"make",
     obj:{ type:"user", ref:"u-bob-1841693218357636549",
     name:"Bob" }}
```

The server announces that object creation for the context is complete by sending the context object a `"ready"` message:

```
A← { to:"ctx-chatroom1", op:"ready" }
```

Finally, the server places Alice into the context and sends a `"make"` message to establish the client presence representing Alice herself (a similar `"make"` message will also be sent to Bob's client, telling him about Alice's arrival, though Alice doesn't see that):

```
A← { to:"ctx-chatroom1", op:"make",
     obj:{ type:"user", ref:"u-alice-5682117796337941936",
     name:"Alice" }, you:true }
```

Compare the `"make"` messages for Alice and Bob: the one for Alice contains an additional parameter, `you`, that indicates to Alice's client that this is the object that represents her rather than anybody else. At the same time, the server will send a separate message on its connection to Bob's client:

```
B← { to:"ctx-chatroom1", op:"make",
     obj:{ type:"user", ref:"u-alice-5682117796337941936",
     name:"Alice" }}
```

This tells Bob that Alice has arrived.

Note also that even though Alice's ref is "`u-alice`" and Bob's is "`u-bob`", what shows up here is somewhat different: "`u-alice-5682117796337941936"` and "`u-bob-1841693218357636549`". These longer refs designate Alice's and Bob's existence in this particular instance. The longer ref serves two purposes: first, if Alice is in more than one context at the same time (some applications allow this), this disambiguates which version of Alice a given message refers to. Second, it prevents other clients who are not in the context from being able to guess how to address messages to her there without the collusion of Alice or somebody else who is already there.

Alice is now in the context, so she says something to those present, using the `"say"` message that we defined above.

```
A→ { to:"ctx-chatroom1", op:"say", speech:"Hi everybody!" }
```

This is echoed to everybody here, including to Alice herself:

```
AB← { to:"ctx-chatroom1", op:"say",
     from:"u-alice-5682117796337941936",
     speech:"Hi everybody!" }
```

Even though Alice knows what she said, echoing the message back to her in this way allows her client to see the sequencing of her `"say"` message in the stream of other actions going on around the same time.

Now Bob replies. Alice doesn't see his message to the server, of course, but she sees the broadcast it generates:

```
AB← { to:"ctx-chatroom1", op:"say",
     from:"u-bob-1841693218357636549",
     speech:"Howdy, Alice!" }
```

Now Bob has to take his leave, so he says good-bye:

```
AB← { to:"ctx-chatroom1", op:"say",
     from:"u-bob-1841693218357636549",
     speech:"Sorry, gotta run. My toast is burning!" }
```

Once Bob leaves the context, the server announces to Alice's client that he is gone by instructing it to delete his presence:

```
A← { to:"u-bob-1841693218357636549", op:"delete" }
```

Just as the convention for telling a client to create an object is to address a `make` message to the new object's container, the convention for telling a client to remove an object is to address a `delete` message to the object itself.

Notice that although the message is transmitted to Alice's client, it is addressed to the client's representation of Bob. This illustrates the point made above that messages are addressed along two dimensions: the client connection along which the message will be sent (in this case, the connection to Alice's client), and the object to which the message should be delivered (in this case, Alice's client presence of Bob).

Finally, Alice herself leaves the context, by sending an `"exit"` message:

```
A→ { to:"ctx-chatroom1", op:"exit" }
```

The `"exit"` message is optional, in that she could simply drop her connection, but doing it this way leaves her in contact with the server, possibly to enter a different context if she so chose.

Adding persistent state

Let's expand our example a bit, by adding another message to the chat protocol. We'll add a `"push"` request, where one client can ask everyone in the room to direct their web browser to a particular web page (this might be used in interactive presentations, for example). Let's make the message a bit more complicated than "`say`" by allowing the sender the option to name a browser frame, with the idea that this is going to be used by some kind of fancy multi-paned browser client:

```
{ to:contextRef, op:"push", url:urlString, frame:optFrame }
```

which the chat mod then echoes like it did the `"say"` message, by attaching the sender's identity:

```
{ to:contextRef, op:"push", url:urlString, frame:optFrame,
  from:pusherRef }
```

Here we've labeled the value of the frame property *`optFrame`*, to connote that it is optional. What we mean is that if the sender did not care to specify a frame, it would omit this property from the message altogether.

Pushing somebody's browser to a different URL is a pretty aggressive thing to do, so we might not want to allow this in every chat room — perhaps the chat rooms intended for casual conversation wouldn't permit this feature, but the ones intended for sales presentations would. So we'll add a property to the mod to enable us to configure this. This means adding some state to our chat mod class, and giving it a different constructor:

```
/** Whether users are permitted to push URLs to other users. */
private boolean amAllowingPush;

@JSONMethod({ "allowpush" })
public SimpleChat(OptBoolean allowPush) {
    amAllowingPush = allowPush.value(false);
```

```
    }
```

Here we have chosen to make the push permission property optional, defaulting to disallowing push if nothing is specified (this means that the behavior of our existing context `"ctx-chatroom1"` will not be changed by adding this new feature). The class `OptBoolean` is provided by the Elko API to denote an optional boolean parameter (the framework also provides obvious similar things like `OptInteger`, `OptString`, and so on). The `OptBoolean` class has a `value()` method that lets you obtain the value of the parameter, while specifying a default to use if the parameter was actually omitted in the message itself. The server's message dispatch system understands about these kinds of optional parameters and invokes the message handler with an instance of `OptBoolean` that encodes whether the value was true or false or absent from the message altogether.

Our extended chat mod also requires an enhanced version of the `encode()` method:

```
public JSONLiteral encode(EncodeControl control) {
    JSONLiteral result = new JSONLiteral("chat", control);
    if (!control.toClient()) {
        result.addParameter("allowpush", amAllowingPush);
    }
    result.finish();
    return result;
}
```

(We consider that the permissions setting is strictly the server's internal business, so we don't pass that information to the client.)

The implementation of the "`push`" message handler is pretty similar to that of the "`say`" handler, except that we check the permission setting:

```
@JSONMethod({ "url", "frame" })
public void push(User from, String url, OptString frame)
    throws MessageHandlerException
{
    if (amAllowingPush) {
        ensureSameContext(from);
        context().send(msgPush(context(), from, url,
                               frame.value(null)));
    } else {
        throw new MessageHandlerException("push not allowed here")
    }
}
```

Note the way the optional `frame` parameter is handled.

The method that generates the outgoing "`push`" message should be relatively obvious as well:

```
static JSONLiteral msgPush(Referenceable target,
                           Referenceable from,
                           String url, String frame)
{
    JSONLiteral msg = new JSONLiteral(target, "push");
    msg.addParameter("from", from);
    msg.addParameter("url", url);
    msg.addParameterOpt("frame", frame);
    msg.finish();
    return msg;
}
```

This looks pretty similar to the `msgSay()` method. The one new wrinkle here is the use of `JSONLiteral`'s `addParameterOpt()` method. This only bothers to actually add the parameter to the message if it has a value other than `null`.

We might want to add another context to our server's configuration:

```
{
    type:"context",
    ref:"ctx-presentationroom1",
    name:"Example Presentation Room",
    capacity:20,
    mods: [
        { type:"chat", allowpush:true }
    ]
}
```

Interaction with this context would look pretty similar to the example given earlier, except that Alice can now send things like:

```
A→ { to:"ctx-presentationroom1", op:"push",
     url:"http://suddenlysocial.net/awesome.html" }
```

Getting Even Fancier

Now let's suppose we wanted to add the ability for users to carry on side conversations: to send messages from one user to another that wouldn't be shared with everybody in the context. You'd think the obvious way for this to work would be for Alice's client to be able to send something like:

```
A→ { to:" u-bob-1841693218357636549", op:"say",
     speech:"Can you believe this guy?" }
```


i.e., the same "`say`" message we've been using, except addressed to a specific user instead of to the context. And if you thought that, you'd be right! But of course as we have things now, this won't work. The message would be delivered to the server presence of Bob, not to the context. It's the context that has our `SimpleChat` mod attached, not the `User` object representing Bob. Bob's user object doesn't yet have any way to know what to do with this message (depending on how the server is configured, it would either reply with a "message not understood" message to Alice's session, or silently discard the message unprocessed). What you'd like is something similar to the `SimpleChat` mod, but for users instead of contexts. The following should do the trick:

```
package com.example.tutorial;

import org.elkoserver.foundation.json.JSONMethod;
import org.elkoserver.foundation.json.MessageHandlerException;
import org.elkoserver.json.EncodeControl;
import org.elkoserver.json.JSONLiteral;
import org.elkoserver.server.context.Mod;
import org.elkoserver.server.context.User;
import org.elkoserver.server.context.UserMod;

/**
 * User mod to let users in a context talk privately
 * to each other.
 */
public class PrivateChat extends Mod implements UserMod {

    @JSONMethod
    public PrivateChat() {
    }

    public JSONLiteral encode(EncodeControl control) {
        if (control.toRepository()) {
            JSONLiteral result =
                new JSONLiteral("privchat", control);
            result.finish();
            return result;
        } else {
            return null;
        }
    }

    @JSONMethod({ "speech" })
    public void say(User from, String speech)
        throws MessageHandlerException
    {
        ensureSameContext(from);
        User who = (User) object();
        JSONLiteral response =
```

```
                SimpleChat.msgSay(who, from, speech);
            who.send(response);
            if (from != who) {
                from.send(response);
            }
        }
    }
```

A few things to note here:

The declaration of the class:

```
    public class PrivateChat extends Mod implements UserMod {
```

looks a lot like the declaration for `SimpleChat`, except that it declares itself to be a `UserMod` rather than a `ContextMod` — a mod that is attached to users rather than to contexts.

The constructor and the `encode` method both look essentially the same as those for SimpleChat before we added the push feature (which we've left out of this class in the interest of brevity).

The big difference is in the `"say"` message handler:

```
    @JSONMethod({ "speech" })
    public void say(User from, String speech)
        throws MessageHandlerException
    {
        ensureSameContext(from);
        User who = (User) object();
        JSONLiteral response =
            SimpleChat.msgSay(who, from, speech);
        who.send(response);
        if (from != who) {
            from.send(response);
        }
    }
```

The method signature looks just like the corresponding handler in the `SimpleChat` mod, which is not surprising since the message protocol is the same. However, there are a couple of things that it does differently. The line:

```
        User who = (User) object();
```

extracts the user to whom this mod is attached. The `object()` method is provided by the `Mod` base class, and always returns the object to which the mod is attached. In this case, it's always a user, so we can safely cast it.

We create the outgoing message with:

```
JSONLiteral response =
    SimpleChat.msgSay(who, from, speech);
```

Note that we just use the message generator method from `SimpleChat` — remember what we said about the possibility of wanting in the future to send the message from someplace else? Well, here it is.

Instead of sending the message out to the context as a whole, we just send it to the client of the person to whom it was addressed, and we also echo it to the sender so that they can see what they did:

```
who.send(response);
if (from != who) {
    from.send(response);
}
```

The `if` statement handles the case where somebody chooses to talk to themselves — hey, I talk to myself all the time — so that in that case they don't get the message echoed back to themselves twice.

Note, by the way, that the value of the `to` property of the "`say`" message that is constructed here is the ref of the user to whom the message was originally addressed, even when it's being echoed back to its original sender who is usually somebody else. Once again, we are distinguishing between the unum to which the message is addressed (i.e., the logical world object that is the target of the message) and the presences to which it is actually delivered (in this case, the particular clients that are concerned with it).

Though we didn't implement push here, the correspondence between a `PrivateChat` version of the `"push"` message handler and its counterpart in the original `SimpleChat` class is directly analogous and so we'll leave adding the push feature as an exercise for the reader, should you so choose.

Just as the context "`ctx-chatroom1`" was defined with an entry in the server's object repository, each user also is defined by such an entry. We'll wave our hands for a moment about how that user's entry got there in the first place (presumably as part of the application's new user registration process), but with this user mod a user's entry would look something like this:

```
{
    type:"user",
    ref:"u-alice",
    name:"Alice",
    mods: [
        { type:"privchat" }
```



```
        ]
    }
```

Other users would look essentially the same, though of course with their own refs and names. This pattern is fine as long as all the contexts in this server include the chat functionality. Chat is generally useful, so it's certainly conceivable that you might want to associate the `chat` mod with every context, but it's also entirely conceivable that you might want to have some kinds of non-chat contexts as well. If you do, then having a `privchat` mod permanently attached to every user might lead to problems, since the private chat functionality would always be enabled, even in contexts where it is inappropriate.

Really what we'd like to do is attach the `privchat` mod to a user when they arrive in a chat context and remove it again when they leave, so that it doesn't get saved as part of the user's persistent state. Fortunately, Elko provides a couple of hooks that let us do just that. Each `Context` keeps a list of `UserWatcher` objects that get notified any time a user enters or exits the context. The `SimpleChat` mod can register just such a callback as part of its initialization using the `Context` class' `registerUserWatcher()` method.

The intuitively natural place for `SimpleChat` to register a user watcher is in its constructor, but unfortunately, that won't work because the mod's constructor is called before the mod is attached to the context (since the mod has to exist before it can be attached!) However, it is not unusual for mods to need to do these kinds of initializations. The framework provides for this by defining an interface called `ObjectCompletionWatcher`.

When the server is constructing a context (or any other unum), it notes any mods that implement `ObjectCompletionWatcher`, and once the construction is complete (meaning everything has been loaded from the repository, and all the mods the context will have have been constructed and attached), it invokes the `objectIsComplete()` method on each of the mods that implements it. The `objectIsComplete()` method can perform any additional initializations that require access to the context or to any of the other attached mods.

Here is how we put these hooks to together to make private chat work the way we want:

First we have the `SimpleChat` mod class implement the `ObjectCompletionWatcher` interface:

```
public class SimpleChat extends Mod
        implements ObjectCompletionWatcher, ContextMod {
```

We'll use the `objectIsComplete()` method to create and install a `UserWatcher`: that will in turn create a `PrivateChat` mod and attach it to each arriving user:

```
public void objectIsComplete() {
    context().registerUserWatcher(
        new UserWatcher() {
            public void noteUserArrival(User who) {
                PrivateChat privateChat =
                    new PrivateChat();
                privateChat.attachTo(who);
            }
            public void noteUserDeparture(User who) { }
        });
}
```

The `UserWatcher` interface defines two methods, one for when users arrive in the context and the other for when they leave. We could have used the `noteUserDeparture()` method to remove the `PrivateChat` mod from departing users, but it is simpler and safer to simply make the `PrivateChat` mod be ephemeral by changing its `encode()` method:

```
public JSONLiteral encode(EncodeControl control) {
    return null;
}
```

This mod is never serialized anywhere — remember that, as we said above, returning `null` from `encode()` means "ignore me". In particular, when a user who has this mod attached is serialized to the repository, this mod gets left behind. Since there's now no serialized form for the `PrivateChat` mod, you might also want to remove the `@JSONMethod` annotation from its constructor.

Beyond Chat

Since this is intended to be a platform for games, lets extend our example application beyond the primitive chat facility and start adding some more game-like (or, at least, virtual world-like) features.

As it stands now, the chat room is an all-or-nothing sort of thing: a user is either in a room or not, but there is no other sense of space, 2-D, 3-D, or otherwise. One of the first things we'd like to do for many kinds of games is add some kind of spatial abstraction. There are a couple of different senses of space that are relevant to think about here: the space formed by different contexts and their relationships to each other, and the space that is interior to a context. The first of these is the source of much of the power and scalability of the Elko platform, but involves a host of subtleties that we're better off saving for later. The second is much more straightforward and visceral, so we'll begin there.

The simplest spatial model is probably a basic two-dimensional Cartesian coordinate space. We can give every user and item in a context an (X,Y) position, which clients can then use to control where to display these objects' visual representations on the screen.

The context server framework provides a very basic positioning mechanism for intra-context positions of just this sort. The basic unum classes all have a `position()` method that returns a position and a `setPosition()` method that enables you to set it:

```
public Position position()
public void setPosition(Position pos)
```

`Position` itself is a very primitive interface (`Positions` have to be `Encodable`, and that's about all). This primitiveness means that you can define your own position class with whatever semantics make sense for your game, without being constrained by arbitrary framework rules. Unfortunately, this also means that the interface as defined doesn't really give you any help in doing this. However, the framework also provides a couple of basic implementations ready made, the simplest of which is `CartesianPosition`, a simple 2-D integer Cartesian coordinate pair that is exactly what we need for this example.

Now all we need is a way for users to have their avatars walk around. Naturally, we do this with a mod, something like this:

```
package com.example.tutorial;

import org.elkoserver.foundation.json.JSONMethod;
import org.elkoserver.foundation.json.MessageHandlerException;
import org.elkoserver.json.EncodeControl;
import org.elkoserver.json.JSONLiteral;
import org.elkoserver.json.Referenceable;
import org.elkoserver.server.context.CartesianPosition;
import org.elkoserver.server.context.Mod;
import org.elkoserver.server.context.Msg;
import org.elkoserver.server.context.User;
import org.elkoserver.server.context.ContextMod;

/**
 * A simple context mod to enable users in a context to move aroun
 */
public class Movement extends Mod implements ContextMod {
    @JSONMethod
    public Movement() {
    }

    public JSONLiteral encode(EncodeControl control) {
        if (control.toClient()) {
            return null;
        } else {
            JSONLiteral result =
                new JSONLiteral("movement", control);
            result.finish();
            return result;
```

```
            }
        }

        @JSONMethod({ "x", "y" })
        public void move(User from, int x, int y)
            throws MessageHandlerException
        {
            ensureSameContext(from);
            from.setPosition(new CartesianPosition(x, y));
            context().send(msgMove(who, x, y));
        }

        static JSONLiteral msgMove(Referenceable who, int x, int y) {
            JSONLiteral msg = new JSONLiteral(who, "move");
            msg.addParameter("x", x);
            msg.addParameter("y", y);
            msg.finish();
            return msg;
        }
    }
```

If you've been following along at home up to this point, most of this should be relatively unsurprising. The only really new element is in the `move()` method where it says:

```
        from.setPosition(new CartesianPosition(x, y));
```

but this does pretty much exactly what you'd expect: it generates a new Cartesian coordinate pair and makes this the user's new position (behind the scenes, it also ensures that this changed position will be saved to the repository when the user is checkpointed).

Also, of course, a lot of things have been simplified here for pedagogical purposes: we don't really do any kind of parameter checking on the (X,Y) position that the user moves to. We allow the user to move anywhere they want; all they have to do is assert their destination. Also, we don't attempt any kind of path analysis or collision detection, and we don't pay any attention at all to the dimension of time — the user moves instantaneously from their old position to their new one. Most real game designs would most likely want to introduce additional mechanisms to address one or more of these complicating elements.

Let's go ahead and introduce one such complicating element, just to show how it's done. Let's give each context a bounding rectangle: a set of minima and maxima for the (X,Y) coordinates, beyond which we'll refuse to let the user go.

You might have noticed an asymmetry in how we implemented the `Movement` mod: it was declared as a context mod, meaning that the user's incoming `"move"` message, requesting movement, is addressed to the context, whereas the server's outgoing `"move"` message, announcing the movement to everyone in the context, is addressed to

the user. The `Movement` mod could instead have been a user mod, with the request being addressed to the user — it is the user who is moving, after all. The only thing that would be different about its implementation in that case would be which interface the class is declared to implement, `UserMod` instead of `ContextMod`. Either choice may be reasonable, depending on other considerations. In this case we chose to make it a context mod because we knew what the next step in this tutorial was going to be. We need someplace to hold the context's position boundary, and a context mod is the how we'd do that. But since the only use of the position boundary information is in checking a movement destination, we'd like the bounds to be readily available to whatever code is performing that check. We could implement a context mod that holds the bounds and a user mod that performs the move, with the user mod getting the bounds information by looking it up from the context (the API supports this), but that's a lot more work for no particular benefit in this case. So the `Movement` mod gets some instance variables, and revised constructor, `encode()` and `move()` methods:

```
private int myMinX;
private int myMinY;
private int myMaxX;
private int myMaxY;

@JSONMethod({"minx", "miny", "maxx", "maxy"})
public Movement(OptInteger minX, OptInteger minY,
                OptInteger maxX, OptInteger maxY)
{
    myMinX = minX.value(-100);
    myMinY = minY.value(-100);
    myMaxX = maxX.value(100);
    myMaxY = maxY.value(100);
}

public JSONLiteral encode(EncodeControl control) {
    if (control.toClient()) {
        return null;
    } else {
        JSONLiteral result =
            new JSONLiteral("movement", control);
        result.addParameter("minx", myMinX);
        result.addParameter("miny", myMinY);
        result.addParameter("maxx", myMaxX);
        result.addParameter("maxy", myMaxY);
        result.finish();
        return result;
    }
}

@JSONMethod({ "x", "y" })
public void move(User from, int x, int y)
    throws MessageHandlerException
```

```
{
    ensureSameContext(from);
    if (x < myMinX || myMaxX < x || y < myMinY || myMaxY < y) {
        from.send(Msg.msgError(object(), "move",
                               "movement out of bounds"));
    } else {
        from.setPosition(new CartesianPosition(x, y));
        context().send(msgMove(from, x, y));
    }
}
```

Once again, we make the various configuration parameters optional, setting the bounds values to some reasonable defaults if they are not otherwise specified. Most of the additions should be obvious, with the possible exception of the updated `move()` method.

One design question that comes up in a situation like this is: what do you do if a parameter check fails? Here we chose to respond with an error message so that we could talk about how we do that, but in general there are a number of different reasonable options. One would simply be to clamp the movement destination at the boundary. Another, the one we chose here, would be to reject the request with some kind of error response. In other cases, it sometimes makes sense to silently reject the request, or even to disconnect the user, if, for example, the only plausible way invalid parameters could find their way in would be as the consequence of somebody inappropriately hacking with their client. Yet another alternative would be to throw a `MessageHandlerException`, which will either silently swallow the error or return an error message, depending on how the server is configured; however, the idea behind a `MessageHandlerException` is that something happened that was not merely wrong, but something that should not have happened. You would not generally use it for something that should be regarded as a "normal" error — in this case, whether an out of bounds destination is normal or not is really a design decision on the part of the developer.

A pattern we like to follow — one that you are not obligated to use, but one that we provide a little bit of support for — is that a normal error response to a request is indicated by a reply message with the same target and message verb and a single parameter named `error` containing a string explaining the problem. So in this case we'd send back:

```
{ to:userRef, op:"move", error:"movement out of bounds" }
```

This is message is produced by the call to `Msg.msgError()`. The `Msg` class is a utility class that provides a number of different message generator methods for common, recurring message patterns such as this. Note that, depending on how an error result is going to be interpreted by the client, this is not always the ideal way to communicate a problem of this sort, but it's a simple convention that works well much of the time. You might not want to use this pattern if there can be multiple requests from the client in flight at once that could possibly generate errors and the client needs a way to associate

an error with the particular request that triggered it. In that case you'd want to also include some kind of message ID or sequence number in both the request and the response. Also, as a rule, the string in the error parameter is intended to be used for debugging, not for display to the human at the controls. Providing meaningful user feedback might warrant the inclusion of additional diagnostic parameters in the response beyond a simple indicator of which flavor of problem occurred.

Now that we have the ability for our users to move around inside the space defined by the context, let's give them something they can do there. If there's an item in the context (on the ground, so to speak), we'd like a user to be able to handle it: to walk over to it, pick it up, carry it someplace else, and then put it down again. A simple, portable item like this is basically an inert prop, but it can act as a bit of set dressing for role playing, and it can be the foundation for more complex types of objects that enable more complicated game play as the implementation evolves.

Lacking any more sophisticated behavior, all these prop items are essentially alike. However, we can label each of them with a tag that says what sort of object it represents, so that the client could distinguish between them, for example by displaying a different graphic for each kind. As with many of the examples in this tutorial, the presentation of an object on the client can be made arbitrarily sophisticated and configurable by the addition of other types of descriptive information. Once again, we'll leave that as an exercise for the reader, but hopefully the generalization from the simple example given here will be reasonably obvious.

So here is the `Prop` mod, the most sophisticated example yet (but still only about 70 lines of code even counting whitespace and the Java boilerplate):

```
package com.example.tutorial;

import org.elkoserver.foundation.json.JSONMethod;
import org.elkoserver.foundation.json.MessageHandlerException;
import org.elkoserver.json.EncodeControl;
import org.elkoserver.json.JSONLiteral;
import org.elkoserver.server.context.CartesianPosition;
import org.elkoserver.server.context.Item;
import org.elkoserver.server.context.ItemMod;
import org.elkoserver.server.context.Mod;
import org.elkoserver.server.context.Msg;
import org.elkoserver.server.context.User;

public class Prop extends Mod implements ItemMod {
    private String myKind;

    private static final int GRAB_DISTANCE = 5;

    public Prop(String kind) {
        myKind = kind;
```

```
}

public JSONLiteral encode(EncodeControl control) {
    JSONLiteral result = new JSONLiteral("prop", control);
    result.addParameter("kind", myKind);
    result.finish();
    return result;
}

@JSONMethod
public void grab(User from) throws MessageHandlerException {
    ensureInContext(from);
    Item item = (Item) object();
    if (!item.isPortable()) {
        throw new MessageHandlerException(
            "attempt to grab non-portable item " + item);
    }

    CartesianPosition itemPos =
        (CartesianPosition) object().position();
    CartesianPosition userPos =
        (CartesianPosition) from.position();
    int dx = itemPos.x() - userPos.x();
    int dy = itemPos.y() - userPos.y();
    if (dx*dx + dy*dy > GRAB_DISTANCE*GRAB_DISTANCE) {
        throw new MessageHandlerException(
            "attempt to grab too far away item " + item);
    }

    item.setContainer(from);
    item.setPosition(null);
    context().sendToNeighbors(from, Msg.msgDelete(item));
    from.send(Movement.msgMove(item, 0, 0, from));
}

@JSONMethod
public void drop(User from) throws MessageHandlerException {
    ensureHolding(from);
    Item item = (Item) object();
    if (!item.isPortable()) {
        throw new MessageHandlerException(
            "attempt to drop non-portable item " + item);
    }
    CartesianPosition pos =
        (CartesianPosition) from.position();
    item.setPosition(pos);
    item.setContainer(context());
    item.sendObjectDescription(context().neighbors(from),
```

```
                                        context());
            from.send(Movement.msgMove(item, pos.x(), pos.y(),
                                        context()));
        }
    }
```

This is an item mod. The only piece of state is the `kind` string, which allows the client to tell one species of prop from another. Hopefully by now, the state declaration, serialization, and deserialization should look completely routine to you. All the interesting stuff is in the two message handler methods `grab()` and `drop()`, which enable users to pick things up and put them back down.

Let's look at `grab()` first:

```
    @JSONMethod
    public void grab(User from) throws MessageHandlerException {
```

The `"grab"` message says, basically "grab this". It indicates a desire on the behalf of the sending user to pick up the item to which this mod is attached. To begin with, we do a number of checks to see if the operation is going to be allowed. First, make sure we are actually in the same context with the item (this check is extremely common, which is why we provide its implementation for you).

```
        ensureInContext(from);
```

Next, we want to be sure the item is something that affords being picked up. All items have a *portability* property that indicates whether or not they can be moved around. Since the whole point of the `Prop` mod is to provide such manipulative functions, we might consider that any item with the `Prop` mod attached is portable by definition, but we'd like to leave the option open for the future for `Props` to have other things that can be done to them even if they are stationary, so we'll go ahead with the portability check:

```
        Item item = (Item) object();
        if (!item.isPortable()) {
            throw new MessageHandlerException(
                "attempt to grab non-portable item " + item);
        }
```

As was mentioned earlier, the `object()` method is provided by the `Mod` base class, and always returns the unum to which the mod is attached. In this case, the unum is always an `Item`, so once again we are safe casting it. Then we use the `Item` class' built-in `isPortable()` function to check for portability, and fail if the check fails. Note that we go ahead and use the `MessageHandlerException` here because we expect the client to preflight this operation and not even attempt it on non-portable items — the game UI will be better anyway if grabbing is simply not available on ungrabbable objects, rather than inflicting an error message on the user.

Next we check to see if the item is close enough to where the user is. Here we do a simple Pythagorean theorem distance check based on a fixed reachability distance. A more complex example might parameterize the distance in a more sophisticated way.

```
CartesianPosition itemPos =
    (CartesianPosition) object().position();
CartesianPosition userPos =
    (CartesianPosition) from.position();
int dx = itemPos.x() - userPos.x();
int dy = itemPos.y() - userPos.y();
if (dx*dx + dy*dy > GRAB_DISTANCE*GRAB_DISTANCE) {
    throw new MessageHandlerException(
        "attempt to grab too far away item " + item);
}
```

Once all the checks have been passed, it's time to actually move the item. This has two parts: changing the item's container from the context to the user, and removing its position. (The latter is not strictly necessary, but is more a bit of data hygiene. This way, when the item is inside the user, it isn't carrying bogus position information with some arbitrary coordinate it inherited from when it was on the ground. Note also that other kinds of containers might actually have internal geometry of their own, and position might have some entirely different meaning there. We're not bothering with that in this example though.)

```
item.setContainer(from);
item.setPosition(null);
```

Finally, we need to tell everybody in the context what happened. This also has two parts: informing the user who made the request and informing everybody else. The reason for these two different notifications is that the different users will see different things. From the point of view of the user who issued the grab request, the item was moved from the ground (the context) into the user's inventory (the user as container). From everybody else's point of view, the item has disappeared (once again, this is a design decision: we can allow a container to be *transparent* — contents visible to everyone — but in this case we have chosen to make a user's inventory private).

First, we tell the requesting user:

```
from.send(Movement.msgMove(item, 0, 0, from));
```

Here, we have extended the meaning and syntax of the `"move"` message: in addition to the X and Y position parameters, we now also allow an optional container specifier. The extended meaning is "move this object to position (X,Y) in the given container". If the container is left unspecified, it is assumed that the container is unchanged, i.e., that the object is just moved around within its current container.

Telling everybody else is a little more subtle:

```
        context().sendToNeighbors(from, Msg.msgDelete(item));
```

The `Msg.msgDelete()` method produces a `"delete"` message, another one of the common messages that is used repeatedly in the Elko framework. The `sendToNeighbors()` method is provided by the context, and the pattern may be new to you but we use it a lot. The idea is: send this message to every user in the context except for one specific user who will be excluded. In other words, send this message to all the neighbor users of a given user.

And then we're all done!

The `drop()` method has some similar features, but a few wrinkles of its own:

```
    @JSONMethod
    public void drop(User from) throws MessageHandlerException {
```

As before, we begin by doing some checks. The first check is that the user who is asking to drop the item is actually in possession of it in the first place:

```
        ensureHolding(from);
```

The `ensureHolding()` method is another one of the guard methods provided by the `Mod` base class. It throws an exception if the object to which the mod is attached is not in the containership hierarchy of the user indicated in the parameter.

Next, we once again check to be sure that the item is portable:

```
        Item item = (Item) object();
        if (!item.isPortable()) {
            throw new MessageHandlerException(
                "attempt to drop non-portable item " + item);
        }
```

Note that it is possible for a user to be carrying around a non-portable item. Non-portability simply means is that the item's location can't be changed, but that unchangeable position might be inside the user's inventory. This might be used, for example, for some kind of tool that the user should always have access to.

Next, we set the item's position and container. This looks a lot like `grab()`, except that we copy the user's position (we're dropping the item on the ground at the spot where the user is currently located), and, of course, the new container is the context:

```
        CartesianPosition pos = (CartesianPosition) from.position();
        item.setPosition(pos);
        item.setContainer(context());
```

Finally, we have the same kind of two-part notification. This is the dual to what happened with `grab()`: from the requesting user's point of view, the item is being moved from their inventory to the ground; from everybody else's point of view, the item is coming into existence for the first time:

First, we tell the requesting user:

```
from.send(Movement.msgMove(item, pos.x(), pos.y(),
          context()));
```

this looks pretty much just like it did in `grab()`, except the movement is in the other direction.

Telling everybody else introduces some more new ideas:

```
item.sendObjectDescription(context().neighbors(from),
                           context());
```

The `Item` class' `sendObjectDescription()` method generates and sends a "`make`" message for the item. The two parameters are (1) who to deliver the message to, and (2) what container the item should be created into (this is the unum to which the "`make`" message is targeted). This is different from an ordinary message send operation because the act of sending an item description is a bit more complicated: not only do we need to generate and send a "`make`" message to create the item itself but we may also need additional "`make`" messages for the rest of item's contents (and their contents, and so on) if the item is a container whose contents are visible (what we above called a *transparent container*).

The first parameter to `sendObjectDescription()` needs to be something that implements the `Deliverer` interface. This interface is defined as:

```
public interface Deliverer {
    void send(JSONLiteral message);
}
```

Its `send()` method delivers a message somewhere. You've seen this already: the `User` class implements `Deliverer`: it delivers the message to the client associated with the user. The `Context` class also implements `Deliverer`: it delivers the message to all the clients of all the users in the context. We've used these kinds of `send()` operations already, though you didn't associate them with this interface at the time. The expression:

```
context().neighbors(from)
```

returns a `Deliverer` that delivers to all of the neighbors of the user `from` in the context. It is similar to the `sendToNeighbors()` operation discussed above (in fact, it is implemented internally using the `sendToNeighbors()` method), but is packaged in a way that allows it to be reused for multiple messages, since, as we mentioned,

`sendObjectDescription()` can result in more than one message being generated and sent.

There are lots more bells and whistles in the various classes that the framework provides, but you should now have a grasp of the basics. Explore the Javadoc for the various server classes, especially the ones in the package `org.elkoserver.server.context`, to get a sense of all the different things you can do. And go build something fun!

