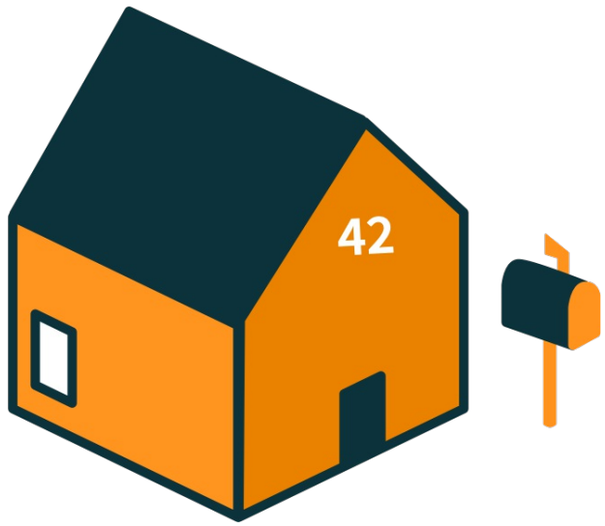
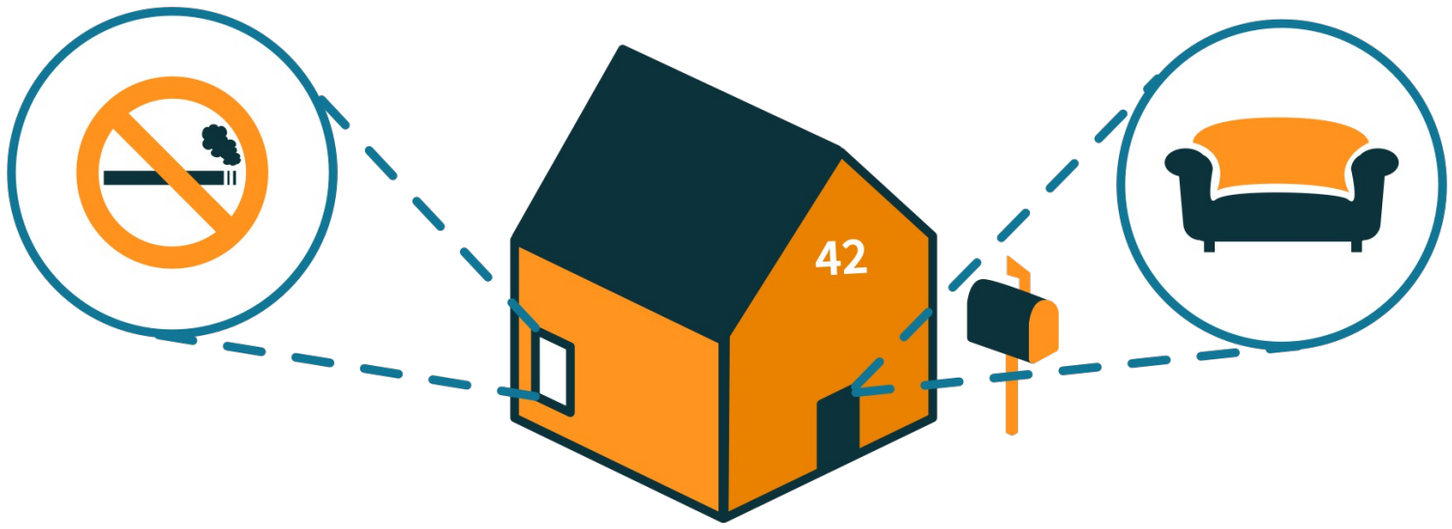
Entities

Entities



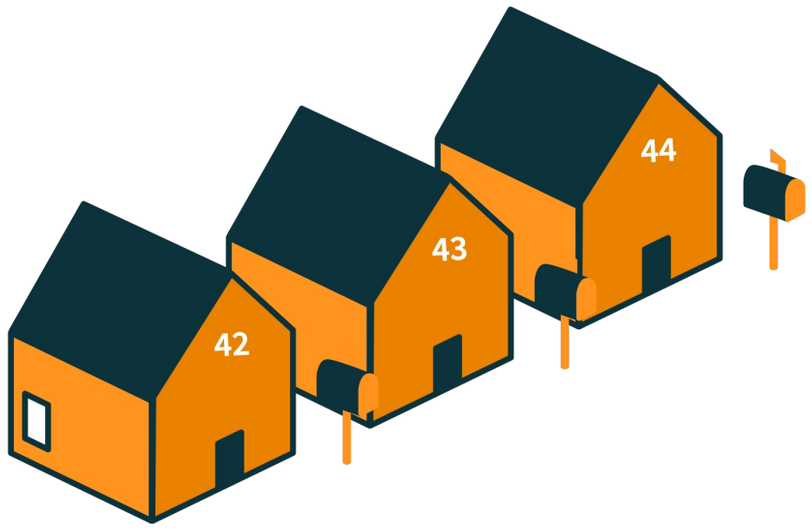
- In our gRPC *Schema* we saw how to define a *Service*.
- In Akka Serverless, our *Service* manages *Entities*.
- The term *Entity* comes from *Domain Driven Design (DDD)* and is used to model domain concepts.
- *Entities* represent specific domain objects such as a Customer, Order, or Reservation.
- Akka Serverless supports different types of *Entities*. We will focus on *EventSourced* entities.

Properties of an Entity



- *Entities* have the following properties.
  - They are uniquely identifiable.
  - They are stateful.
  - They encapsulate domain logic.

Uniquely Identifiable

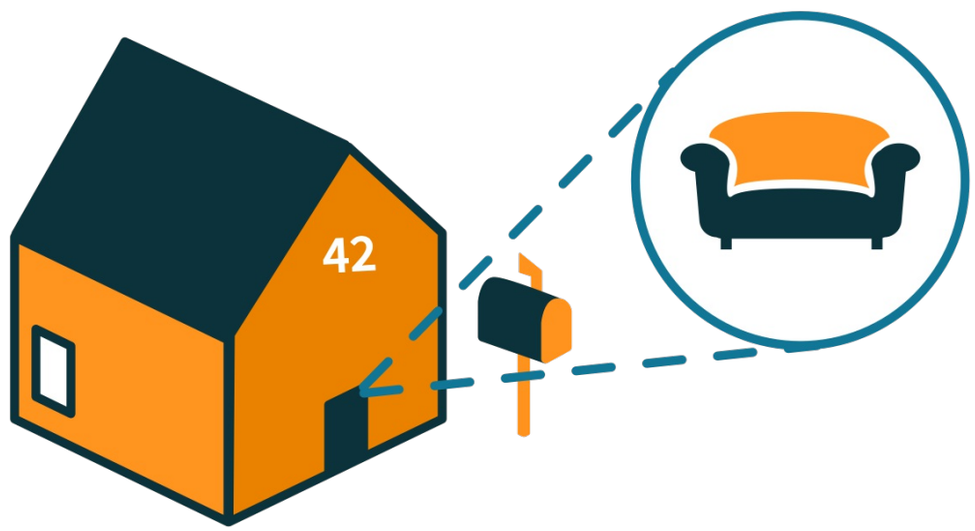


- All *Entities* must be uniquely identifiable using an *Entity Key*.
- This *Entity Key* is immutable. It cannot change.
- If you change the *Entity Key* then you are no longer referring to the same *Entity*.

Choosing your Entity Key

- To ensure good distribution of your *Entities*, the *Entity Key* must meet the following:
  - It must be unique.
  - It must have a reasonably high number of possible values (cardinality).
  - It should avoid *hotspots* (ideally).
- Hotspots can occur when some *Entities* receive significantly more traffic than others.
- *Compound Keys* are possible.

Stateful



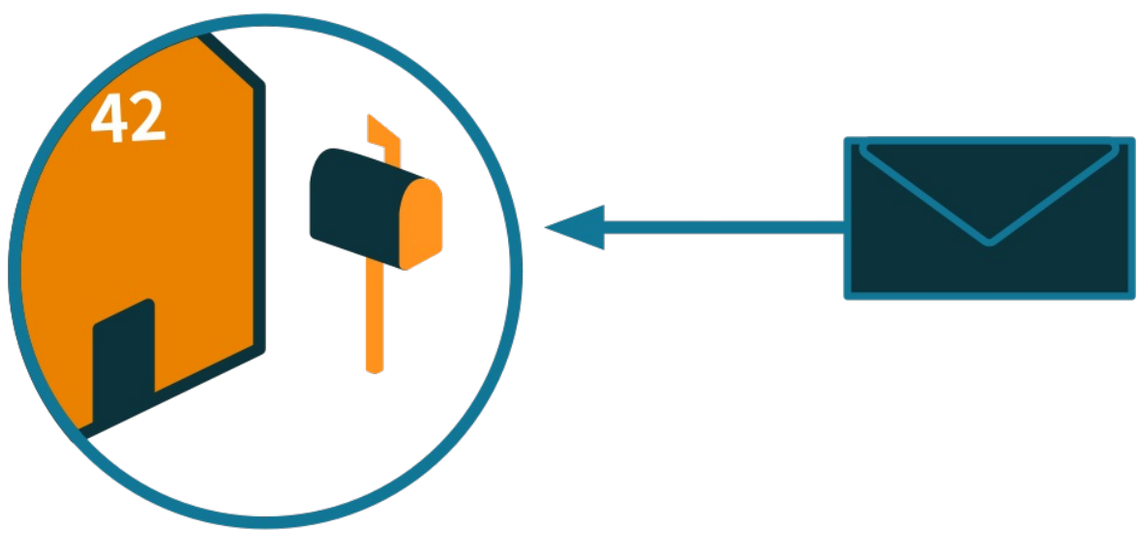
- *Entities* are stateful.
- They contain the *State* associated with the domain concept that they are modelling.
- As long as the *Entity Key* remains the same, the *State* it encapsulates is free to change.
- The *State* of the *Entity* is therefore mutable.

Encapsulating Domain Logic

- The *Entity* also encapsulates any domain logic associated with it.
- It will contain the business rules that are necessary to manipulate the state of the *Entity*.

Consistency in Entities

Sending Messages

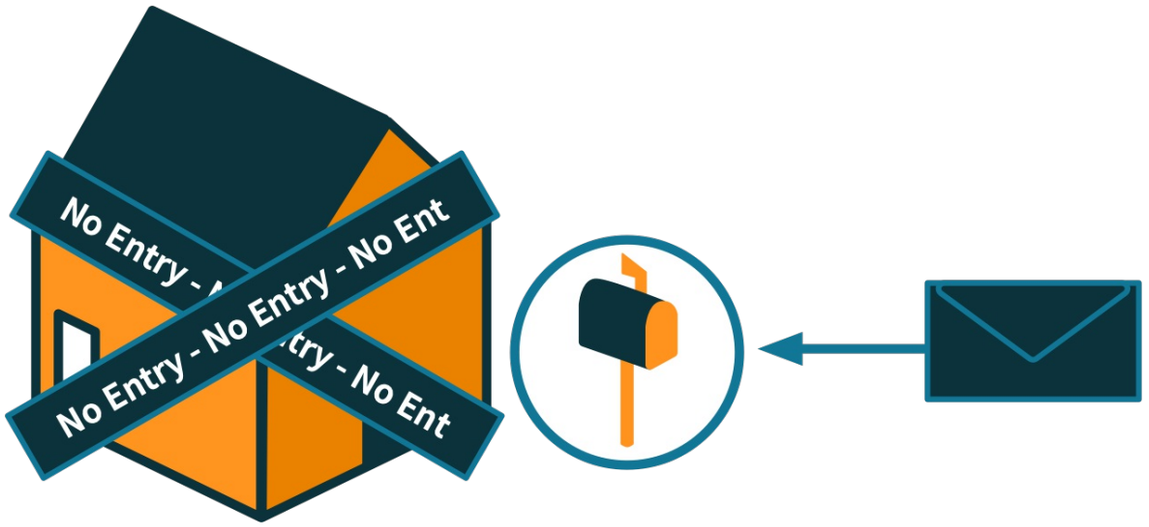


- Requests are addressed to an *Entity* using the *Entity Key*.
- Akka Serverless will create an instance of the *Entity* based on the *Key* (if required).
- That instance will handle all requests addressed to the *Entity*.
- If necessary, Akka Serverless can redistribute *Entities* to other machines.

Concurrency

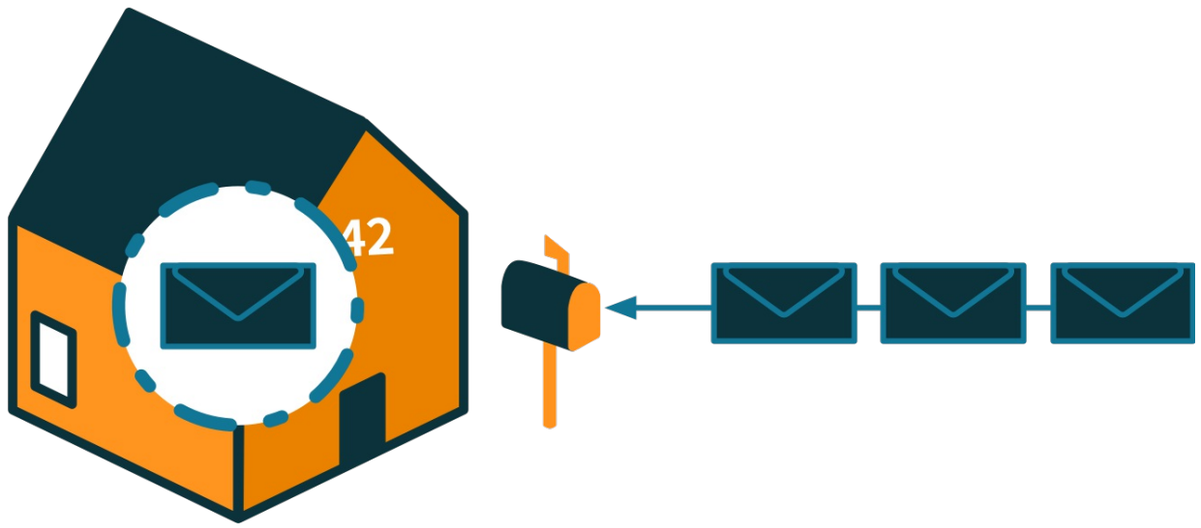
- Each *Entity* processes one message at a time.
- There is no concurrency within the *Entity*.
- However, many *Entities* can operate in parallel.
- Therefore, the *Entity* is the unit of concurrency within the *Service*.

State Ownership



- Each *Entity* manages its own *State*.
- Access to the *state* requires sending the *Entity* a message or command.
- No backdoor access is permitted.
  - No shared *State*, no shared database tables.
- This creates isolation between *Entities*.

Consistency



- Each *Entity* only processes one message at a time.
- Each *Entity* manages its own *State*.
- This means that there is no concurrent access to the *State*.
- This allows the *Entity* to act as a consistency boundary for the *State*.
- In essence, we can guarantee that the *Entity* will always be strongly consistent.

Implementing Entities

Event Sourced Entities

- .proto

```
package lightbend.example;

service MyService { ... }
```

- .js

```
const { EventSourced } = require('cloudstate').EventSourced;

const entity = new EventSourced(
  ['file1.proto', 'file2.proto'],
  'lightbend.example.MyService',
  { /* Options */ }
);
```

- To create an *EventSourced* entity we need the following:
  - An array of proto files defining the schema for our *Service*.
  - A fully qualified name for the service (as defined by the *Schema*).
  - A set of options to further configure the *Entity*.
- Note: We will go into more detail on *EventSourced* entities later in the course.

Defining the State

```
message State {
  string field = 1;
}
```

- Remember, *Entities* are stateful.
- They contain mutable *State*, associated with a unique *Key*.
- This *State* needs to be contained in a message, defined by the *Schema*.
- Note: Empty messages can be used if required.

```
message Empty {
}
```

Setting the Initial State

```
const State = entity.lookupType('lightbend.example.State');
entity.setInitial((userId) => State.create({ field: 'value' }));
```

- When each instance of the entity is created, we need to provide an *initial* state.
- Given a *Key* we will map that to an initial state.
- As additional messages are processed, this state will be mutated.
- Note: You will need to lookup the approriate type, in order to use it.

Command Handlers

```
// Schema
service MyService {
  rpc CommandHandler(Command) returns (Response) {...}
}
```

- As messages are sent to an *Entity* they are processed by a *Handler*.
- *Handlers* are defined in our gRPC *Schema* as a method.
- The method will later be mapped to a function in our *Service Implementation*.
- In Akka Serverless we refer to these functions as *Command Handlers*.

Implementing Command Handlers

- .proto

```
rpc CommandHandler(Command) returns (Response) {...}
```

- .js

```
entity.myCommandHandler = function (command, state, context) {
  ...
  return response;
};
```

- A *Command Handler* is implemented as a function with three parameters:
  - command - The incoming command (as defined by the gRPC *Schema*).
  - state - The current state, prior to executing the command.
  - context - A context object containing some helpful functions/information (more on this later).
- The command handler should return a response that will be sent back to the caller (as defined by the gRPC *Schema*).

Behaviors

- .proto

```
rpc CommandHandler(Command) returns (Response) {...}
```

- .js

```
entity.setBehavior((state) => ({
  return {
    commandHandlers: {
      CommandHandler: entity.myCommandHandler,
    },
  };
}));
```

- *Command Handlers* are grouped together into a *Behavior*.
- The *Behavior* provides a map from the methods in our *Schema* to the JavaScript implementation.
- When the method in the *Schema* is called, the JavaScript function will be executed.

Exporting the Entity

```
module.exports = entity;
```

- Once the entity is ready, it needs to be exported.
- This will make it available so that it can be accessed by other parts of the application.

Starting the Server

- The final step is to start the server (typically inside index.js).

```
require('./myentity').start();
```

- This will start an Akka Serverless server that only hosts a single entity type.
- Alternatively, we can create a server that hosts multiple entity types.

```
const entityType1 = require("./entityType1")
const entityType2 = require("./entityType2")

const server = new cloudstate.CloudState();
server.addEntity(entityType1);
server.addEntity(entityType2);
server.start();
```