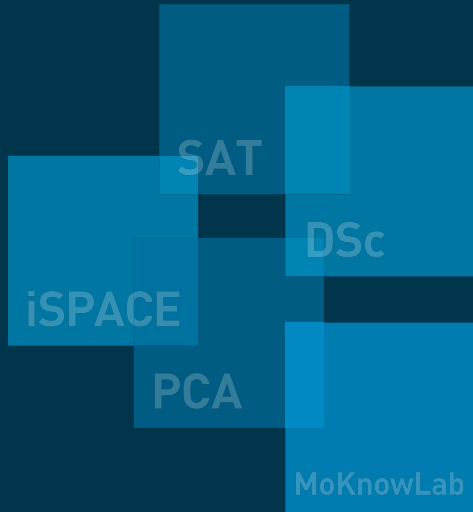
RSAFG

Research Studio **SAT**

Introduction: Bot Framework



Online-Resources

- Framework Documentation:
  https://github.com/researchstudio-sat/webofneeds
  -> webofneeds -> won-bot
- Bot Template: https://github.com/researchstudio-sat/bot-skeleton
- Examplebots:
  - https://github.com/researchstudio-sat/won-debugbot
  - https://github.com/researchstudio-sat/won-spoco

What‘s a won-bot?

- An autonomous entity interacting with atoms on the Web of Needs
- Able to do about the same things as a (human) user
  - Creating atoms
  - Creating connections
  - Writing messages
- Usually a spring boot app using an in-memory database

Interaction with Atoms

- Generally good to keep track of:
  - known atoms
  - owned atoms
  - active connections

Take care not to connect bots to each other to avoid endless loops!

Bot Context

- Stores information and known objects
- May be used as a cache for atom information
- Can store additional information about atoms
- Access via `BotContextWrapper`

Code Example – Creating an Atom

```
EventListenerContext ctx = getEventListenerContext();
DefaultAtomModelWrapper atomModelWrapper = new DefaultAtomModelWrapper();
WonNodeInformationService wonNodeInformationService =
                                    ctx.getWonNodeInformationService();

URI wonNodeUri = ctx.getNodeURISource().getNodeURI();
URI atomURI = wonNodeInformationService.generateAtomURI(wonNodeUri);

// add information to the atom model
atomModelWrapper = new DefaultAtomModelWrapper(atomURI);
atomModelWrapper.setTitle("your title");
atomModelWrapper.setDescription("your description");
...
```

Code Example – Creating an Atom

```
// add sockets for connections between atoms
atomModelWrapper.addSocket(atomURI.toString() + "#socket0",        SocketType.ChatSocket.getURI());
atomModelWrapper.addSocket(atomURI.toString() + "#socket1",        SocketType.HoldableSocket.getURI());

// prepare the creation message that will be sent to the node
Dataset atomDataset = atomModelWrapper.copyDataset();
WonMessage createAtomMessage = createWonMessage(wonNodeInformationService,                                                                    atomURI,
wonNodeUri, atomDataset);

// remember the atom URI so we can react to success/failure responses
EventBotActionUtils.rememberInList(ctx, atomURI, uriListName);
```

Code Example – Creating an Atom

```
EventListener successCallback = eventS -> {
                // things to happen after successful creation
};
EventListener failureCallback = eventF -> {
                // things to happen in case of failure
};

// register event listeners for callback methods
EventBotActionUtils.makeAndSubscribeResponseListener(createAtomMessage, successCallback, failureCallback, ctx);

// send creation message to node
getEventListenerContext().getWonMessageSender().sendWonMessage(createAtomMessage);
```

Events, Actions, Behaviours

- Bots are event-driven
- Events are sent as signals that something happened
- Define listeners that trigger specific actions in response to events

- For example: use an `AtomCreatedEvent` to trigger a `SayHelloAction`

Events, Actions, Behaviours

- Events/Actions can only be created and deleted, not paused
- Behaviours act as a wrapper to event listeners and actions
- Behaviours can be activated and deactivated easily
- Behaviours can be seen as small modules containing possible bot interactions that can be added and removed

Code Example - Behaviour

```
public HelloBehaviour(EventListenerContext context, String name) {
super(context, name);
}

protected void onActivate(Optional<Object> message) {
Action sayHelloAction = new Action(context);
subscribeWithAutoCleanup(ActEvent.class, new ActionOnEventListener(context,
sayHelloAction, 1));
}

protected void onDeactivate(Optional<Object> message) { ... }
```

Extensions

- Define additional bot features
- Usually consist of at least an interface and a behaviour
- Examples: `MatcherExtension`, `ServiceAtomExtension`
- For things that are nice to have but not needed in every bot