Spring Statemachine - Reference Documentation

1.0.2.RC1

Janne Valkealahti Pivotal

Table of Contents

Preface

Concept of a state machine is most likely older that any of a reader of this reference documentation and definitely older than a Java language itself. Description of finite automate dates back to 1943 when gentlemens Warren McCulloch and Walter Pitts wrote a paper about it. Later George H. Mealy presented a state machine concept in 1955 which is known as a Mealy Machine. A year later in 1956 Edward F. Moore presented another paper which is known as a Moore Machine. If you're ever read anything about state machines, names Mealy and Moore should have popped up at some point.

This reference documentations contains following parts.

Part I, "Introduction" introduction to this reference documentation

Part III, "Using Spring Statemachine" describes the usage of Spring State Machine(SSM)

Part V, "State Machine Examples" more detailed state machine samples

Part VI, "FAQ" frequently ask questions

Part VII, "Appendices" generic info about used material and state machines

Part I. Introduction

Spring Statemachine(SSM) is a framework for application developers to use traditional state machine concepts with Spring applications. SSM aims to provide following features:

- Easy to use flat one level state machine for simple use cases.
- Hierarchical state machine structure to ease complex state configuration.
- State machine regions to provide even more complex state configurations.
- Usage of triggers, transitions, guards and actions.
- Type safe configuration adapter.
- State machine event listeners.
- Spring IOC integration to associate beans with a state machine.

Before you continue it’s worth to go through appendices Section B.2, “Glossary” and Section B.3, “A State Machines Crash Course” to get a generic idea of what state machines are mostly because rest of a documentation expects reader to be fairly familiar with state machine concepts.

1. Background

State machines are powerful because behaviour is always guaranteed to be consistent and relatively easily debugged due to ways how operational rules are written in stone when machine is started. Idea is that your application is and may exist in a finite number of states and then something happens which takes your application from one state to the next. What will drive a state machine are triggers which are either based on events or timers.

It is much easier to design high level logic outside of your application and then interact with a state machine with a various different ways. You can simply interact with a state machine by sending event, listening what a state machine does or simply request a current state.

Traditionally state machines are added to a existing project when developer realizes that code base is starting to look like a plate full of spaghetti. Spaghetti code looks like never ending hierarchical structure of IFs, ELSEs and BREAK clauses and probably compiler should ask developer to go home when things are starting to look too complex.

2. Usage Scenarios

Project is a good candidate to use state machine if:

- Application or part of its structure can be represented as states.
- You want to split complex logic into smaller manageable tasks.
- Application is already suffering concurrency issues with i.e. something happening asynchronously.

You are already trying to implement a state machine if:

- Use of boolean flags or enums to model situations.
- Having variables which only have meaning for some part of your application lifecycle.
- Looping through if/else structure and checking if particular flag or enum is set and then making further exceptions what to do when certain combination of your flags and enums exists or doesn't exist together.

Part II. Getting started

If you’re just getting started with Spring Statemachine, this is the section for you! Here we answer the basic “what?”, “how?” and “why?” questions. You’ll find a gentle introduction to Spring Statemachine. We’ll then build our first Spring Statemachine application, discussing some core principles as we go.

3. System Requirements

Spring Statemachine 1.0.2.RC1 is built and tested with JDK 8(all artifacts have JDK 7 compatibility) and Spring Framework 4.2.2.RELEASE and doesn’t require any other dependencies outside of Spring Framework within its core system.

Other optional parts like Chapter 23, *Using Distributed States* has dependencies to a `Zookeeper`, while Part V, “State Machine Examples” has dependencies to spring-shell and spring-boot which pulls other dependencies beyond framework itself.

4. Modules

The following modules are available for Spring Statemachine.

Module	Description
spring-statemachine-core	Core system of a Spring Statemachine.
spring-statemachine-recipes-common	Common recipes which doesn’t require dependencies outside of a core framework.
spring-statemachine-zookeeper	`Zookeeper` integration for a distributed state machine.
spring-statemachine-test	Support module for state machine testing.

5. Using Gradle

Here is a typical `build.gradle` file:

```
buildscript {
    repositories {
        maven { url "http://repo.spring.io/libs-release" }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.2.5.RELEASE")
    }
}

apply plugin: 'base'
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'
version =  '0.1.0'
archivesBaseName = 'gs-statemachine'

repositories {
    mavenCentral()
    maven { url "http://repo.spring.io/libs-release" }
    maven { url "http://repo.spring.io/libs-milestone" }
    maven { url "http://repo.spring.io/libs-snapshot" }
}

dependencies {
    compile("org.springframework.statemachine:spring-statemachine-core:1.0.0.BUILD-SNAPSHOT")
    compile("org.springframework.boot:spring-boot-starter:1.2.5.RELEASE")
    testCompile("org.springframework.statemachine:spring-statemachine-test:1.0.0.BUILD-SNAPSHOT")
}

task wrapper(type: Wrapper) {
    gradleVersion = '1.11'
}
```

Note

Replace `1.0.0.BUILD-SNAPSHOT` with a version you want to use.

Having a normal project structure you'd build this with command:

```
# ./gradlew clean build
```

Expected Spring Boot packaged fat-jar would be `build/libs/gs-statemachine-0.1.0.jar`.

Note

You don't need repos `libs-milestone` and `libs-snapshot` for production development.

6. Using Maven

Here is a typical `pom.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.springframework</groupId>
    <artifactId>gs-statemachine</artifactId>
    <version>0.1.0</version>
    <packaging>jar</packaging>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.2.5.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.statemachine</groupId>
            <artifactId>spring-statemachine-core</artifactId>
            <version>1.0.0.BUILD-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
            <version>1.0.0.BUILD-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.statemachine</groupId>
            <artifactId>spring-statemachine-test</artifactId>
            <version>1.0.0.BUILD-SNAPSHOT</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <artifactId>maven-failsafe-plugin</artifactId>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>integration-test</goal>
                            <goal>verify</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

    <repositories>
        <repository>
            <id>spring-release</id>
            <url>http://repo.spring.io/libs-release</url>
            <snapshots><enabled>false</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestone</id>
            <url>http://repo.spring.io/libs-milestone</url>
            <snapshots><enabled>false</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-snapshot</id>
            <url>http://repo.spring.io/libs-snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
    </repositories>
```

> **Note**
>
> Replace `1.0.0.BUILD-SNAPSHOT` with a version you want to use.

Having a normal project structure you’d build this with command:

```
# mvn clean package
```

Expected Spring Boot packaged fat-jar would be `target/gs-statemachine-0.1.0.jar`.

> **Note**
>
> You don’t need repos `libs-milestone` and `libs-snapshot` for production development.

7. Developing your first Spring Statemachine application

Let’s start by creating a simple Spring Boot `Application` class implementing `CommandLineRunner`.

```
@SpringBootApplication
public class Application implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

Add states and events:

```
public enum States {
    SI, S1, S2
}

public enum Events {
    E1, E2
}
```

Add state machine configuration:

```
@Configuration
@EnableStateMachine
public class StateMachineConfig
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineConfigurationConfigurer<States, Events> config)
            throws Exception {
        config
            .withConfiguration()
                .autoStartup(true)
                .listener(listener());
    }

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.SI)
                    .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.SI).target(States.S1).event(Events.E1)
                .and()
            .withExternal()
                .source(States.S1).target(States.S2).event(Events.E2);
    }

    @Bean
    public StateMachineListener<States, Events> listener() {
        return new StateMachineListenerAdapter<States, Events>() {
            @Override
            public void stateChanged(State<States, Events> from, State<States, Events> to) {
                System.out.println("State change to " + to.getId());
            }
        };
    }
}
```

Implement `CommandLineRunner`, autowire `StateMachine`:

```
@Autowired
private StateMachine<States, Events> stateMachine;

@Override
public void run(String... args) throws Exception {
    stateMachine.sendEvent(Events.E1);
    stateMachine.sendEvent(Events.E2);
}
```

Depending whether you build your application using `Gradle` or `Maven` it’s run `java -jar build/libs/gs-statemachine-0.1.0.jar` or `java -jar target/gs-statemachine-0.1.0.jar` respectively.

What is expected for running this command is a normal Spring Boot output but if you look closely you see lines:

```
State change to SI
State change to S1
State change to S2
```

Part III. Using Spring Statemachine

This part of the reference documentation explains the core functionality that Spring Statemachine provides to any Spring based application.

Chapter 8, *Statemachine Configuration* the generic configuration support.

Chapter 9, *State Machine Factories* the generic state machine factory support.

Chapter 12, *Using Actions* the actions support.

Chapter 13, *Using Guards* the guard support.

Chapter 14, *Using Extended State* the extended state support.

Chapter 15, *Using StateContext* the state context support.

Chapter 16, *Triggering Transitions* the use of triggers.

Chapter 17, *Listening State Machine Events* the use of state machine listeners.

Chapter 18, *Context Integration* the generic Spring application context support.

Chapter 19, *State Machine Accessor* the state machine internal accessor support.

Chapter 20, *State Machine Interceptor* the state machine error handling support.

Chapter 21, *State Machine Error Handling* the state machine interceptor support.

Chapter 22, *Persisting State Machine* the state machine persisting support.

Chapter 23, *Using Distributed States* the distributed state machine support.

Chapter 24, *Testing Support* the state machine testing support.

8. Statemachine Configuration

One of the common tasks when using a Statemachine is to design its runtime configuration. This chapter will focus on how Spring Statemachine is configured and how it leverages Spring's lightweight IoC containers to simplify the application internals to make it more manageable.

> **Note**
>
> Configuration examples in this section are not feature complete, i.e. you always need to have definitions of both states and transitions, otherwise state machine configuration would be ill-formed. We have simply made code snippets less verbose by leaving other needed parts away.

8.1 Using *enable* annotations

We use familiar spring *enabler* annotations to ease configuration. Two annotations exists, *@EnableStateMachine* and *@EnableStateMachineFactory*. These annontations if placed in a *@Configuration* class will enable some basic functionality needed by a state machines.

@EnableStateMachine is used when a configuration wants to create an instance of a *StateMachine*. Usually *@Configuration* class extends adapters `EnumStateMachineConfigurerAdapter` or `StateMachineConfigurerAdapter` which allows user to override configuration callback methods. We automatically detect if user is using these adapter classes and modify runtime configuration logic.

@EnableStateMachineFactory is used when a configuration wants to create an instance of a *StateMachineFactory*.

> **Note**
>
> Usage examples of these are shown in below sections.

8.2 Configuring States

We'll get into more complex configuration examples a bit later but let's first start with a something simple. For most simple state machine you just use `EnumStateMachineConfigurerAdapter` and define possible states, choose initial and optional end state.

```
@Configuration
@EnableStateMachine
public class Config1Enums
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

}
```

It's also possible to use strings instead of enums as states and events by using `StateMachineConfigurerAdapter` as shown below. Most of a configuration examples is using enums but generally speaking strings and enums can be just interchanged.

```
@Configuration
@EnableStateMachine
public class Config1Strings
        extends StateMachineConfigurerAdapter<String, String> {

    @Override
    public void configure(StateMachineStateConfigurer<String, String> states)
            throws Exception {
        states
            .withStates()
                .initial("S1")
                .end("SF")
                .states(new HashSet<String>(Arrays.asList("S1","S2","S3","S4")));
    }

}
```

Note

Using enums will bring more safe set of states and event types but limits possible combinations to compile time. Strings don't have this limitation and allows user to use more dynamic ways to build state machine configurations but doesn't allow same level of safety.

8.3 Configuring Hierarchical States

Hierarchical states can be defined by using multiple `withStates()` calls where `parent()` can be used to indicate that these particular states are sub-states of some other state.

```
@Configuration
@EnableStateMachine
public class Config2
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .state(States.S1)
                .and()
                .withStates()
                    .parent(States.S1)
                    .initial(States.S2)
                    .state(States.S2);
    }

}
```

8.4 Configuring Regions

There are no special configuration methods to mark a collection of states to be part of an orthogonal state. To put it simple, orthogonal state is created when same hierarchical state machine has multiple set of states each having a initial state. Because an individual state machine can only have one initial state, multiple initial states must mean that a specific state must have multiple independent regions.

```
@Configuration
@EnableStateMachine
public class Config10
        extends EnumStateMachineConfigurerAdapter<States2, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States2, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States2.S1)
                .state(States2.S2)
                .and()
                .withStates()
                    .parent(States2.S2)
                    .initial(States2.S2I)
                    .state(States2.S21)
                    .end(States2.S2F)
                    .and()
                .withStates()
                    .parent(States2.S2)
                    .initial(States2.S3I)
                    .state(States2.S31)
                    .end(States2.S3F);
    }

}
```

8.5 Configuring Transitions

We support three different types of transitions, `external`, `internal` and `local`. Transitions are either triggered by a signal which is an event sent into a state machine or a timer.

```
@Configuration
@EnableStateMachine
public class Config3
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.S1).target(States.S2)
                .event(Events.E1)
                .and()
            .withInternal()
                .source(States.S2)
                .event(Events.E2)
                .and()
            .withLocal()
                .source(States.S2).target(States.S3)
                .event(Events.E3);
    }

}
```

8.6 Configuring Guards

Guards are used to protect state transitions. Interface *Guard* is used to do an evaluation where method has access to a *StateContext*.

```
@Configuration
@EnableStateMachine
public class Config4
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.S1).target(States.S2)
                .event(Events.E1)
                .guard(guard())
                .and()
            .withExternal()
                .source(States.S2).target(States.S3)
                .event(Events.E2)
                .guardExpression("true");

    }

    @Bean
    public Guard<States, Events> guard() {
        return new Guard<States, Events>() {

            @Override
            public boolean evaluate(StateContext<States, Events> context) {
                return true;
            }
        };
    }

}
```

In above two different types of guard configurations are used. Firstly a simple *Guard* is created as a bean and attached to transition between states `S1` and `S2`.

Secondly a simple SPeL expression can be used as a guard where expression must return a `BOOLEAN` value. Behind a scenes this expression based guard is a *SpelExpressionGuard*. This was attached to transition between states `S2` and `S3`. Both guard in above sample always evaluate to true.

8.7 Configuring Actions

Actions can be defined to be executed with transitions and states itself. Action is always executed as a result of a transition which originates from a trigger.

```
@Configuration
@EnableStateMachine
public class Config51
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.S1)
                .target(States.S2)
                .event(Events.E1)
                .action(action());
    }

    @Bean
    public Action<States, Events> action() {
        return new Action<States, Events>() {

            @Override
            public void execute(StateContext<States, Events> context) {
                // do something
            }
        };
    }

}
```

In above a single `Action` is defined as bean `action` and associated with a transition from `S1` to `S2`.

```
@Configuration
@EnableStateMachine
public class Config52
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1, action())
                .state(States.S1, action(), null)
                .state(States.S2, null, action())
                .state(States.S3, action(), action());
    }

    @Bean
    public Action<States, Events> action() {
        return new Action<States, Events>() {

            @Override
            public void execute(StateContext<States, Events> context) {
                // do something
            }
        };
    }

}
```

Note

Usually you would not define same `Action` instance for different stages but we did it here not to make too much noise in a code snippet.

In above a single `Action` is defined as bean `action` and associated with states `S1`, `S2` and `S3`. There is more going on there which needs more clarification:

- We defined action for initial state `S1`.
- We defined entry action for state `S1` and left exit action empty.
- We defined exit action for state `S2` and left entry action empty.
- We defined entry action as well as exit action for state `S3`.
- Notice how state `S1` is used twice with `initial()` and `state()` functions. This is only needed if you want to define entry or exit actions with initial state.

Important

Defining action with `initial()` function only executes particular action when state machine or sub state is started. Think this action to be initializing action which is only executed once. Action defined with `state()` is then executed if state machine is transitioning back and forward between initial and non-inital states.

8.8 Configuring Pseudo States

Pseudo state configuration is usually done by configuring states and transitions. Pseudo states are automatically added to state machine as states.

Initial State

Simply mark a particular state as initial state by using `initial()` method. There are two methods where one takes extra argument to define an initial action. This initial action is good for example initialize extended state variables.

```
@Configuration
@EnableStateMachine
public class Config11
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1, initialAction())
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

    @Bean
    public Action<States, Events> initialAction() {
        return new Action<States, Events>() {

            @Override
            public void execute(StateContext<States, Events> context) {
                // do something initially
            }
        };
    }

}
```

Terminate State

Simply mark a particular state as end state by using `end()` method. This can be done max one time per individual sub-machine or region.

```
@Configuration
@EnableStateMachine
public class Config1Enums
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

}
```

History State

History state can be defined once for each individual state machine. You need to choose its state identifier and `History.SHALLOW` or `History.DEEP` respectively.

```
@Configuration
@EnableStateMachine
public class Config12
        extends EnumStateMachineConfigurerAdapter<States3, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States3, Events> states)
            throws Exception {
        states
        .withStates()
            .initial(States3.S1)
            .state(States3.S2)
            .and()
            .withStates()
                .parent(States3.S2)
                .initial(States3.S2I)
                .state(States3.S21)
                .state(States3.S22)
                .history(States3.SH, History.SHALLOW);
    }

}
```

Choice State

Choice needs to be defined in both states and transitions to work properly. Mark particular state as choice state by using `choice()` method. This state needs to match source state when transition is configured for this choice.

Transition is configured using `withChoice()` where you define source state and `first/then/last` structure which is equivalent to normal `if/elseif/else`. With `first` and `then` you can specify a guard just like you'd use a condition with `if/elseif` clauses.

Transition needs to be able to exist so make sure `last` is used. Otherwise configuration is ill-formed.

```
@Configuration
@EnableStateMachine
public class Config13
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.SI)
                .choice(States.S1)
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withChoice()
                .source(States.S1)
                .first(States.S2, s2Guard())
                .then(States.S3, s3Guard())
                .last(States.S4);
    }

    @Bean
    public Guard<States, Events> s2Guard() {
        return new Guard<States, Events>() {

            @Override
            public boolean evaluate(StateContext<States, Events> context) {
                return false;
            }
        };
    }

    @Bean
    public Guard<States, Events> s3Guard() {
        return new Guard<States, Events>() {

            @Override
            public boolean evaluate(StateContext<States, Events> context) {
                return true;
            }
        };
    }

}
```

Fork State

Fork needs to be defined in both states and transitions to work properly. Mark particular state as choice state by using `fork()` method. This state needs to match source state when transition is configured for this fork.

Target state needs to be a super state or immediate states in regions. Using a super state as target will take all regions into initial states. Targeting individual state give more controlled entry into regions.

```
@Configuration
@EnableStateMachine
public class Config14
        extends EnumStateMachineConfigurerAdapter<States2, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States2, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States2.S1)
                .fork(States2.S2)
                .state(States2.S3)
                .and()
                .withStates()
                    .parent(States2.S3)
                    .initial(States2.S2I)
                    .state(States2.S21)
                    .state(States2.S22)
                    .end(States2.S2F)
                    .and()
                .withStates()
                    .parent(States2.S3)
                    .initial(States2.S3I)
                    .state(States2.S31)
                    .state(States2.S32)
                    .end(States2.S3F);
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States2, Events> transitions)
            throws Exception {
        transitions
            .withFork()
                .source(States2.S2)
                .target(States2.S22)
                .target(States2.S32);
    }

}
```

Join State

Join needs to be defined in both states and transitions to work properly. Mark particular state as choice state by using `join()` method. This state doesn't need to match either source states or target state in a transition configuration.

Select one target state where transition goes when all source states has been joined. If you use state hosting regions as source, end states of a regions are used as joins. Otherwise you can pick any states from a regions.

```
@Configuration
@EnableStateMachine
public class Config15
        extends EnumStateMachineConfigurerAdapter<States2, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States2, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States2.S1)
                .state(States2.S3)
                .join(States2.S4)
                .and()
                .withStates()
                    .parent(States2.S3)
                    .initial(States2.S2I)
                    .state(States2.S21)
                    .state(States2.S22)
                    .end(States2.S2F)
                    .and()
                .withStates()
                    .parent(States2.S3)
                    .initial(States2.S3I)
                    .state(States2.S31)
                    .state(States2.S32)
                    .end(States2.S3F);
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States2, Events> transitions)
            throws Exception {
        transitions
            .withJoin()
                .source(States2.S2F)
                .source(States2.S3F)
                .target(States2.S5);
    }

}
```

8.9 Configuring Common Settings

Some of a common state machine configuration can be set via a `ConfigurationConfigurer`. This allows to set `BeanFactory`, `TaskExecutor`, `TaskScheduler`, autostart flag for a state machine and register `StateMachineListener` instances.

```
@Configuration
@EnableStateMachine
public class Config17
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineConfigurationConfigurer<States, Events> config)
            throws Exception {
        config
            .withConfiguration()
                .autoStartup(true)
                .beanFactory(new StaticListableBeanFactory())
                .taskExecutor(new SyncTaskExecutor())
                .taskScheduler(new ConcurrentTaskScheduler())
                .listener(new StateMachineListenerAdapter<States, Events>());
    }

}
```

State machine `autoStartup` flag is disabled by default because all instances handling sub-states are controlled by a state machine itself and cannot be started automatically. Also it is much safer to leave this decision to a user whether a machine should be started automatically or not. This flag will only control an autostart of a top-level state machine.

Setting a `BeanFactory`, `TaskExecutor` or `TaskScheduler` exist for conveniance for a user and are also use within a framework itself.

Registering `StateMachineListener` instances is also partly for convenience but is required if user wants to catch callback during a state machine lifecycle like getting notified of a state machine start/ stop events. Naturally it is not possible to listen a state machine start events if `autoStartup` is enabled unless listener can be registered during a configuration phase.

`DistributedStateMachine` is configured via `withDistributed()` which allows to set a `StateMachineEnsemble` which if exists automatically wraps created `StateMachine` with `DistributedStateMachine` and enables distributed mode.

```
@Configuration
@EnableStateMachine
public class Config18
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineConfigurationConfigurer<States, Events> config)
            throws Exception {
        config
            .withDistributed()
                .ensemble(stateMachineEnsemble());
    }

    @Bean
    public StateMachineEnsemble<States, Events> stateMachineEnsemble()
            throws Exception {
        // naturally not null but should return ensemble instance
        return null;
    }

}
```

More about distributed states, refer to section Chapter 23, *Using Distributed States*.

9. State Machine Factories

There are use cases when state machine needs to be created dynamically instead of defining static configuration at compile time. For example if there are custom components which are using its own state machines and these components are created dynamically it is impossible to have a static state machined build during the application start. Internally state machines are always build via a factory interfaces and this then gives user an option to use this feature programmatically. Configuration for state machine factory is exactly same as you've seen in various examples in this document where state machine configuration is hard coded.

9.1 Factory via Adapter

Actually creating a state machine using *@EnableStateMachine* will work via factory so *@EnableStateMachineFactory* is merely exposing that factory via its interface.

```
@Configuration
@EnableStateMachineFactory
public class Config6
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.S1)
                .end(States.SF)
                .states(EnumSet.allOf(States.class));
    }

}
```

Now that you've used *@EnableStateMachineFactory* to create a factory instead of a state machine bean, it can be injected and used as is to request new state machines.

```
public class Bean3 {

    @Autowired
    StateMachineFactory<States, Events> factory;

    void method() {
        StateMachine<States,Events> stateMachine = factory.getStateMachine();
        stateMachine.start();
    }
}
```

Adapter Factory Limitations

Current limitation of factory is that all actions and guard it is associating with created state machine will share a same instances. This means that from your actions and guard you will need to specifically handle a case that same bean will be called by a different state machines. This limitation is something which will be resolved in future releases.

9.2 State Machine via Builder

Using adapters shown above has a limitation imposed by its requirement to work via Spring `@Configuration` classes and application context. While this is a very clear model to configure a state machine instances it will limit configuration at a compile time which is not always what a user wants to do. If there is a requirement to build more dynamic state machines, a simple builder pattern can be used to construct similar instances. Using strings as states and events this builder pattern can be used to build fully dynamic state machines outside of a Spring application context as shown above.

```
StateMachine<String, String> buildMachine1() throws Exception {
    Builder<String, String> builder = StateMachineBuilder.builder();
    builder.configureStates()
        .withStates()
            .initial("S1")
            .end("SF")
            .states(new HashSet<String>(Arrays.asList("S1","S2","S3","S4")));
    return builder.build();
}
```

Builder is using same configuration interfaces behind the scenes that the `@Configuration` model using adapter classes. Same model goes to configuring transitions, states and common configuration via builder's methods. This simply means that whatever you can use with a normal `EnumStateMachineConfigurerAdapter` or `StateMachineConfigurerAdapter` can be used dynamically via a builder.

Note

Currently `builder.configureStates()`, `builder.configureTransitions()` and `builder.configureConfiguration()` interface methods cannot be chained together meaning builder methods needs to be called individually.

```
StateMachine<String, String> buildMachine2() throws Exception {
    Builder<String, String> builder = StateMachineBuilder.builder();
    builder.configureConfiguration()
        .withConfiguration()
            .autoStartup(false)
            .beanFactory(null)
            .taskExecutor(null)
            .taskScheduler(null)
            .listener(null);
    return builder.build();
}
```

It is important to understand on what cases common configuration needs to be used with a machines instantiated from a builder. Configurer returned from a `withConfiguration()` can be used to setup *autoStart*, *TaskScheduler*, *TaskExecutor*, *BeanFactory* and additionally register a *StateMachineListener*. If *StateMachine* instance returned from a builder is registered as a bean via `@Bean`, i.e. *BeanFactory* is attached automatically and then a default *TaskExecutor* can be found from there. If instances are used outside of a spring application context these methods must be used to setup needed facilities.

10. Using Deferred Events

When en event is sent it may fire an `EventTrigger` which then may cause a transition to happen if a state machine is in a state where trigger is evaluated successfully. Normally this may lead to a situation where an event is not accepted and is dropped. However it may be desirable to postpone this event until a state machine enters other state, in which it is possible to accept that event. In other words an event simply arrives at an inconvenient time.

Spring Statemachine provides a mechanism for deferring events for later processing. Every state can have a list of deferred events. If an event in the current state’s deferred event list occurs, the event will be saved (deferred) for future processing until a state is entered that does not list the event in its deferred event list. When such a state is entered, the state machine will automatically recall any saved events that are no longer deferred and will then either consume or discard these events. It is possible for a superstate to have a transition defined on an event that is deferred by a substate. Following same hierarchical state machines concepts, the substate takes precedence over the superstate, the event will be deferred and the transition for the superstate will not be executed. With orthogonal regions where one orthogonal region defers an event and another accepts the event, the accept takes precedence and the event is consumed and not deferred.

The most obvious use case for event deferring is when an event is causing a transition into a particular state and state machine is then returned back to its original state where second event should cause a same transition. Lets take this with a simple example.

```java
@Configuration
@EnableStateMachine
static class Config5 extends StateMachineConfigurerAdapter<String, String> {

    @Override
    public void configure(StateMachineStateConfigurer<String, String> states)
            throws Exception {
        states
            .withStates()
                .initial("READY")
                .state("DEPLOYPREPARE", "DEPLOY")
                .state("DEPLOYEXECUTE", "DEPLOY");
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<String, String> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source("READY").target("DEPLOYPREPARE")
                .event("DEPLOY")
                .and()
            .withExternal()
                .source("DEPLOYPREPARE").target("DEPLOYEXECUTE")
                .and()
            .withExternal()
                .source("DEPLOYEXECUTE").target("READY");
    }
}
```

In above state machine has state *READY* which indicates that machine is ready to process events which would take it into a *DEPLOY* state where the actual deployment would happen. After deploy actions has been executed machine is then returned back into a *READY* state. Sending multiple events in a *READY* state is not causing any trouble if machine is using synchronous executor because event sending would block between event calls. However if executor is using threads then other events may get lost because machine is no longer in a state where event could be processed. Thus deferring some of these events allows machine to preserve these events.

```
@Configuration
@EnableStateMachine
static class Config6 extends StateMachineConfigurerAdapter<String, String> {

    @Override
    public void configure(StateMachineStateConfigurer<String, String> states)
            throws Exception {
        states
            .withStates()
                .initial("READY")
                .state("DEPLOY", "DEPLOY")
                .state("DONE")
                .and()
                .withStates()
                    .parent("DEPLOY")
                    .initial("DEPLOYPREPARE")
                    .state("DEPLOYPREPARE", "DONE")
                    .state("DEPLOYEXECUTE");
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<String, String> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source("READY").target("DEPLOY")
                .event("DEPLOY")
                .and()
            .withExternal()
                .source("DEPLOYPREPARE").target("DEPLOYEXECUTE")
                .and()
            .withExternal()
                .source("DEPLOYEXECUTE").target("READY")
                .and()
            .withExternal()
                .source("READY").target("DONE")
                .event("DONE")
                .and()
            .withExternal()
                .source("DEPLOY").target("DONE")
                .event("DONE");
    }
}
```

In above state machine which is using nested states instead of a flat state model, event *DEPLOY* can be deferred directly in a substate. It is also showing concept of deferring event *DONE* in one of a sub-states which would then override anonymous transition between *DEPLOY* and *DONE* states if state machine happens to be in a *DEPLOYPREPARE* state when *DONE* event is dispatched. In *DEPLOYEXECUTE* state *DONE* event is not deferred, thus event would be handled in a super state.

11. Using Scopes

Support for scopes in a state machine is very limited but it is possible to enable use of *session* scope using a normal spring `@Scope` annotation. Firstly if state machine is build manually via a builder and returned into context as `@Bean`, and secondly via an contifuration adapter. Both of these simply needs an a `@Scope` to be present where *scopeName* is set to *session* and *proxyMode* to `ScopedProxyMode.TARGET_CLASS`. Examples for both use cases are shown below.

```
@Configuration
public class Config3 {

    @Bean
    @Scope(scopeName="session", proxyMode=ScopedProxyMode.TARGET_CLASS)
    StateMachine<String, String> stateMachine() throws Exception {
        Builder<String, String> builder = StateMachineBuilder.builder();
        builder.configureConfiguration()
            .withConfiguration()
                .autoStartup(true)
                .taskExecutor(new SyncTaskExecutor());
        builder.configureStates()
            .withStates()
                .initial("S1")
                .state("S2");
        builder.configureTransitions()
            .withExternal()
                .source("S1")
                .target("S2")
                .event("E1");
        StateMachine<String, String> stateMachine = builder.build();
        return stateMachine;
    }

}
```

```
@Configuration
@EnableStateMachine
@Scope(scopeName="session", proxyMode=ScopedProxyMode.TARGET_CLASS)
public static class Config4 extends StateMachineConfigurerAdapter<String, String> {

    @Override
    public void configure(StateMachineConfigurationConfigurer<String, String> config) throws Exception {
        config
            .withConfiguration()
                .autoStartup(true);
    }

    @Override
    public void configure(StateMachineStateConfigurer<String, String> states) throws Exception {
        states
            .withStates()
                .initial("S1")
                .state("S2");
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<String, String> transitions) throws Exception
 {
        transitions
            .withExternal()
                .source("S1")
                .target("S2")
                .event("E1");
    }

}
```

Once you have scoped state machine into `session`, autowiring it into a `@Controller` will give new state machine instance per session. State machine is then destroyed when `HttpSession` is invalidated.

```
@Controller
public class StateMachineController {

    @Autowired
    StateMachine<String, String> stateMachine;

    @RequestMapping(path="/state", method=RequestMethod.POST)
    public HttpEntity<Void> setState(@RequestParam("event") String event) {
        stateMachine.sendEvent(event);
        return new ResponseEntity<Void>(HttpStatus.ACCEPTED);
    }

    @RequestMapping(path="/state", method=RequestMethod.GET)
    @ResponseBody
    public String getState() {
        return stateMachine.getState().getId();
    }
}
```

Note

Using state machines in a `session` scopes needs a careful planning mostly because it is a relatively heavy component.

12. Using Actions

Actions are one of the most useful components from user perspective to interact and collaborate with a state machine. Actions can be executed in various places in a state machine and its states lifecycle like entering or exiting states or during a transitions.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.SI)
            .state(States.S1, action1(), action2())
            .state(States.S2, action1(), action2())
            .state(States.S3, action1(), action3());
}
```

Above `action1` and `action2` beans are attached to states entry and exit respectively.

```
@Bean
public Action<States, Events> action1() {
    return new Action<States, Events>() {

        @Override
        public void execute(StateContext<States, Events> context) {
        }
    };
}

@Bean
public BaseAction action2() {
    return new BaseAction();
}

@Bean
public SpelAction action3() {
    ExpressionParser parser = new SpelExpressionParser();
    return new SpelAction(
            parser.parseExpression(
                    "stateMachine.sendEvent(T(org.springframework.statemachine.docs.Events).E1)"));
}

public class BaseAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
    }
}

public class SpelAction extends SpelExpressionAction<States, Events> {

    public SpelAction(Expression expression) {
        super(expression);
    }
}
```

You can directly implement *Action* as an anonymous function or create a your own implementation and define appropriate implementation as a bean.

In `action3` a SpEL expression is used to send event **Events.E1** into a state machine.

Note

StateContext is described in section Chapter 15, *Using StateContext*.

12.1 SpEL Expressions with Actions

It is also possible to use SpEL expressions as a replacement for a full *Action* implementation.

13. Using Guards

Above `guard1` and `guard2` beans are attached to states entry and exit respectively.

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.SI).target(States.S1)
            .event(Events.E1)
            .guard(guard1())
            .and()
        .withExternal()
            .source(States.S1).target(States.S2)
            .event(Events.E1)
            .guard(guard2())
            .and()
        .withExternal()
            .source(States.S2).target(States.S3)
            .event(Events.E2)
            .guardExpression("extendedState.variables.get('myvar')");
}
```

You can directly implement *Guard* as an anonymous function or create a your own implementation and define appropriate implementation as a bean. In above sample `guardExpression` is simply checking if extended state variable `myvar` evaluates to *TRUE*.

```
@Bean
public Guard<States, Events> guard1() {
    return new Guard<States, Events>() {

        @Override
        public boolean evaluate(StateContext<States, Events> context) {
            return true;
        }
    };
}

@Bean
public BaseGuard guard2() {
    return new BaseGuard();
}

public class BaseGuard implements Guard<States, Events> {

    @Override
    public boolean evaluate(StateContext<States, Events> context) {
        return false;
    }
}
```

Note

StateContext is described in section Chapter 15, *Using StateContext*.

13.1 SpEL Expressions with Guards

It is also possible to use SpEL expressions as a replacement for a full *Guard* implementation. Only requirement is that expression needs to return a **Boolean** value to satisfy *Guard* implementation. This is demonstrated with a *guardExpression()* function which takes an expression as an argument.

14. Using Extended State

Let’s assume that we’d need to create a state machine tracking how many times a user is pressing a key on a keyboard and then terminate when keys are pressed 1000 times. Possible but a really naive solution would be to create a new state for each 1000 key presses. Going even worse combinations you might suddenly have astronomical number of states which naturally is not very practical.

This is where extended state variables comes into rescue by not having a necessity to add more states to drive state machine changes, instead a simple variable change can be done during a transition.

`StateMachine` has a method `getExtendedState()` which returns an interface `ExtendedState` which gives an access to extended state variables. You can access variables directly via a state machine or `StateContext` during a callback from actions or transitions.

```
public Action<String, String> myVariableAction() {
    return new Action<String, String>() {

        @Override
        public void execute(StateContext<String, String> context) {
            context.getExtendedState()
                .getVariables().put("mykey", "myvalue");
        }
    };
}
```

If there is a need to get notified for extended state variable changes, there are two options; either use `StateMachineListener` and listen `extendedStateChanged(key, value)` callbacks:

```
public class ExtendedStateVariableListener
        extends StateMachineListenerAdapter<String, String> {

    @Override
    public void extendedStateChanged(Object key, Object value) {
        // do something with changed variable
    }
}
```

Or implement a Spring Application context listeners for `OnExtendedStateChanged`. Naturally as mentioned in Chapter 17, *Listening State Machine Events* you can also listen all `StateMachineEvent` events.

```
public class ExtendedStateVariableEventListener
        implements ApplicationListener<OnExtendedStateChanged> {

    @Override
    public void onApplicationEvent(OnExtendedStateChanged event) {
        // do something with changed variable
    }
}
```

15. Using StateContext

StateContext is a domain object representing a current status of a state machine within a transition or an action. Context gives an access to a various information like event, message headers, extended state variables, current transition and a top-level state machine in case there is a need to send events to a further processing.

16. Triggering Transitions

Driving a statemachine is done via transitions which are triggered by triggers. Currently supported triggers are *EventTrigger* and *TimerTrigger*.

16.1 EventTrigger

EventTrigger is the most useful trigger because it allows user to directly interact with a state machine by sending events to it. These events are also called signals. Trigger is added to a transition simply by associating a state to it during a configuration.

```
@Autowired
StateMachine<States, Events> stateMachine;

void signalMachine() {
    stateMachine.sendEvent(Events.E1);

    Message<Events> message = MessageBuilder
            .withPayload(Events.E2)
            .setHeader("foo", "bar")
            .build();
    stateMachine.sendEvent(message);
}
```

In above example we send an event using two different ways. Firstly we simply sent a type safe event using state machine api method `sendEvent(E event)`. Secondly we send event wrapped in a Spring messaging *Message* using api method `sendEvent(Message<E> message)` with a custom event headers. This allows user to add arbitrary extra information with an event which is then visible to *StateContext* when for example user is implementing actions.

16.2 TimerTrigger

TimerTrigger is useful when something needs to be triggered automatically without any user interaction. `Trigger` is added to a transition by associating a timer with it during a configuration.

```
@Configuration
@EnableStateMachine
public class Config2 extends StateMachineConfigurerAdapter<String, String> {

    @Override
    public void configure(StateMachineStateConfigurer<String, String> states)
            throws Exception {
        states
            .withStates()
                .initial("S1")
                .state("S2");
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<String, String> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source("S1")
                .target("S2")
                .event("E1")
                .and()
            .withInternal()
                .source("S2")
                .action(timerAction())
                .timer(1000);
    }

    @Bean
    public TimerAction timerAction() {
        return new TimerAction();
    }
}

public class TimerAction implements Action<String, String> {

    @Override
    public void execute(StateContext<String, String> context) {
        // do something in every 1 sec
    }
}
```

In above we have two states, S1 and S2. We have a normal external transition from S1 to S2 with event E1 but interesting part is when we define internal transition with source state S2 and associate it with Action bean timerAction and timer value of 1000ms. Once a state machine receive event E1 it does a transition from S1 to S2 and timer kicks in. As long as state is kept in S2 TimerTrigger executes and causes a transition associated with that state which in this case is the internal transition which has the timerAction defined.

17. Listening State Machine Events

There are use cases where you just want to know what is happening with a state machine, react to something or simply get logging for debugging purposes. SSM provides interfaces for adding listeners which then gives an option to get callback when various state changes, actions, etc are happening.

You basically have two options, either to listen Spring application context events or directly attach listener to a state machine. Both of these basically will provide same information where one is producing events as event classes and other producing callbacks via a listener interface. Both of these have pros and cons which will be discussed later.

17.1 Application Context Events

Application context events classes are *OnTransitionStartEvent*, *OnTransitionEvent*, *OnTransitionEndEvent*, *OnStateExitEvent*, *OnStateEntryEvent*, *OnStateChangedEvent*, *OnStateMachineStart* and *OnStateMachineStop* and others which extends base event class *StateMachineEvent* These can be used as is with spring typed *ApplicationListener*.

StateMachine will send context events via *StateMachineEventPublisher* it’s set. Default implementation is automatically created if *@Configuration* class is annotated with *@EnableStateMachine*.

```
public class StateMachineApplicationEventListener
        implements ApplicationListener<StateMachineEvent> {

    @Override
    public void onApplicationEvent(StateMachineEvent event) {
    }
}

@Configuration
public class ListenerConfig {

    @Bean
    public StateMachineApplicationEventListener contextListener() {
        return new StateMachineApplicationEventListener();
    }
}
```

Context events are also automatically enabled via *@EnableStateMachine* with machine builder *StateMachine* registered as a bean as shown below.

```
@Configuration
@EnableStateMachine
public class ManualBuilderConfig {

    @Bean
    public StateMachine<String, String> stateMachine() throws Exception {

        Builder<String, String> builder = StateMachineBuilder.builder();
        builder.configureStates()
            .withStates()
                .initial("S1")
                .state("S2");
        builder.configureTransitions()
            .withExternal()
                .source("S1")
                .target("S2")
                .event("E1");
        return builder.build();
    }
}
```

17.2 State Machine Listener

Using *StateMachineListener* you can either extend it and implement all callback methods or use *StateMachineListenerAdapter* class which contains stub method implementations and choose which ones to override.

```
public class StateMachineEventListener
        extends StateMachineListenerAdapter<States, Events> {

    @Override
    public void stateChanged(State<States, Events> from, State<States, Events> to) {
    }

    @Override
    public void stateEntered(State<States, Events> state) {
    }

    @Override
    public void stateExited(State<States, Events> state) {
    }

    @Override
    public void transition(Transition<States, Events> transition) {
    }

    @Override
    public void transitionStarted(Transition<States, Events> transition) {
    }

    @Override
    public void transitionEnded(Transition<States, Events> transition) {
    }

    @Override
    public void stateMachineStarted(StateMachine<States, Events> stateMachine) {
    }

    @Override
    public void stateMachineStopped(StateMachine<States, Events> stateMachine) {
    }

    @Override
    public void eventNotAccepted(Message<Events> event) {
    }

    @Override
    public void extendedStateChanged(Object key, Object value) {
    }

    @Override
    public void stateMachineError(StateMachine<States, Events> stateMachine, Exception exception) {
    }
}
```

In above example we simply created our own listener class *StateMachineEventListener* which extends *StateMachineListenerAdapter*.

Once you have your own listener defined, it can be registered into a state machine via its interface as shown below. It’s just a matter of flavour if it’s hooked up within a spring configuration or done manually at any time of application life-cycle.

```
public class Config7 {

    @Autowired
    StateMachine<States, Events> stateMachine;

    @Bean
    public StateMachineEventListener stateMachineEventListener() {
        StateMachineEventListener listener = new StateMachineEventListener();
        stateMachine.addStateListener(listener);
        return listener;
    }

}
```

17.3 Limitations and Problems

Spring application context is not a fastest eventbus out there so it is advised to give some thought what is a rate of events state machine is sending. For better performance it may be better to use *StateMachineListener* interface. For this specific reason it is possible to use `contextEvents` flag with *@EnableStateMachine* and *@EnableStateMachineFactory* to disable Spring application context events as shown above.

```
@Configuration
@EnableStateMachine(contextEvents = false)
public class Config8
        extends EnumStateMachineConfigurerAdapter<States, Events> {
}

@Configuration
@EnableStateMachineFactory(contextEvents = false)
public class Config9
        extends EnumStateMachineConfigurerAdapter<States, Events> {
}
```

18. Context Integration

It is a little limited to do interaction with a state machine by either listening its events or using actions with states and transitions. Time to time this approach would be too limited and verbose to create interaction with the application a state machine is working with. For this specific use case we have made a spring style context integration which easily attach state machine functionality into your beans.

18.1 Annotation Support

@WithStateMachine annotation can be used to associate a state machine with a existing bean. Within this annotation a property's *source* and *target* can be used to qualify a transition. If *source* and *target* is left empty then any transition is matched.

```
@WithStateMachine
public class Bean1 {

    @OnTransition(source = "S1", target = "S2")
    public void fromS1ToS2() {
    }

    @OnTransition
    public void anyTransition() {
    }
}
```

Default *@OnTransition* annotation can't be used with a state and event enums user have created due to java language limitations, thus string representation have to be used.

Additionally it is possible to access `Event  Headers` and `ExtendedState` by adding needed arguments to a method. Method is then called automatically with these arguments.

```
@WithStateMachine
public class Bean4 {

    @StatesOnTransition(source = States.S1, target = States.S2)
    public void fromS1ToS2(@EventHeaders Map<String, Object> headers, ExtendedState extendedState) {
    }
}
```

However if you want to have a type safe annotation it is possible to create a new annotation and use *@OnTransition* as meta annotation. This user level annotation can make a reference to actual states and events enums and framework will try to match these in a same way.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@OnTransition
public @interface StatesOnTransition {

    States[] source() default {};

    States[] target() default {};
}
```

Above we created a *@StatesOnTransition* annotation which defines `source` and `target` as a type safe manner.

```
@WithStateMachine
public class Bean2 {

    @StatesOnTransition(source = States.S1, target = States.S2)
    public void fromS1ToS2() {
    }
}
```

In your own bean you can then use this *@StatesOnTransition* as is and use type safe `source` and `target`.

19. State Machine Accessor

`StateMachine` is a main interface to communicate with a state machine itself. Time to time there is a need to get more dynamical and programmatic access to internal structures of a state machine and its nested machines and regions. For these use cases a `StateMachine` is exposing a functional interface `StateMachineAccessor` which provides an interface to get access to individual `StateMachine` and `Region` instances.

`StateMachineFunction` is a simple functional interface which allows to apply `StateMachineAccess` interface into a state machine. With jdk7 these will create a little verbose code but with jdk8 lambdas things look relatively non-verbose.

Method `doWithAllRegions` gives access to all `Region` instances in a state machine.

```
stateMachine.getStateMachineAccessor().doWithAllRegions(new
 StateMachineFunction<StateMachineAccess<String,String>>() {

    @Override
    public void apply(StateMachineAccess<String, String> function) {
        function.setRelay(stateMachine);
    }
});

stateMachine.getStateMachineAccessor()
    .doWithAllRegions(access -> access.setRelay(stateMachine));
```

Method `doWithRegion` gives access to single `Region` instance in a state machine.

```
stateMachine.getStateMachineAccessor().doWithRegion(new
 StateMachineFunction<StateMachineAccess<String,String>>() {

    @Override
    public void apply(StateMachineAccess<String, String> function) {
        function.setRelay(stateMachine);
    }
});

stateMachine.getStateMachineAccessor()
    .doWithRegion(access -> access.setRelay(stateMachine));
```

Method `withAllRegions` gives access to all `Region` instances in a state machine.

```
for (StateMachineAccess<String, String> access :
 stateMachine.getStateMachineAccessor().withAllRegions()) {
    access.setRelay(stateMachine);
}

stateMachine.getStateMachineAccessor().withAllRegions()
    .stream().forEach(access -> access.setRelay(stateMachine));
```

Method `withRegion` gives access to single `Region` instance in a state machine.

```
stateMachine.getStateMachineAccessor()
    .withRegion().setRelay(stateMachine);
```

20. State Machine Interceptor

Instead of using a `StateMachineListener` interface one option is to use a `StateMachineInterceptor`. One conceptual difference is that an interceptor can be used to intercept and stop a current state change or transition logic. Instead of implementing full interface, adapter class `StateMachineInterceptorAdapter` can be used to override default no-op methods.

Note

There is one recipe Chapter 25, *Persist* and one sample Chapter 32, *Persist* which are related to use of an interceptor.

Interceptor can be registered via `StateMachineAccessor`. Concept of an interceptor is relatively deep internal feature and thus is not exposed directly via `StateMachine` interface.

```
stateMachine.getStateMachineAccessor()
    .withRegion().addStateMachineInterceptor(new StateMachineInterceptor<String, String>() {

        @Override
        public StateContext<String, String> preTransition(StateContext<String, String> stateContext) {
            return stateContext;
        }

        @Override
        public void preStateChange(State<String, String> state, Message<String> message,
                Transition<String, String> transition, StateMachine<String, String> stateMachine) {
        }

        @Override
        public StateContext<String, String> postTransition(StateContext<String, String> stateContext) {
            return stateContext;
        }

        @Override
        public void postStateChange(State<String, String> state, Message<String> message,
                Transition<String, String> transition, StateMachine<String, String> stateMachine) {
        }

        @Override
        public Exception stateMachineError(StateMachine<String, String> stateMachine,
                Exception exception) {
            return exception;
        }
    });
```

Note

More about error handling shown in above example, see section Chapter 21, *State Machine Error Handling*.

21. State Machine Error Handling

If state machine detects an internal error during a state transition logic it may throw an exception. Before this exception is processed internally, user is given a chance to intercept.

Normal `StateMachineInterceptor` can be used to intercept errors and example of it is shown above.

```
StateMachine<String, String> stateMachine;

void addInterceptor() {
    stateMachine.getStateMachineAccessor()
        .doWithRegion(new StateMachineFunction<StateMachineAccess<String, String>>() {

        @Override
        public void apply(StateMachineAccess<String, String> function) {
            function.addStateMachineInterceptor(
                    new StateMachineInterceptorAdapter<String, String>() {
                @Override
                public Exception stateMachineError(StateMachine<String, String> stateMachine,
                        Exception exception) {
                    // return null indicating handled error
                    return exception;
                }
            });
        }
    });

}
```

When errors are detected, normal event notify mechanism is executed. This allows to use either `StateMachineListener` or Spring Application context event listener, more about these read section Chapter 17, *Listening State Machine Events*.

Having said that, a simple listener would look like:

```
public class ErrorStateMachineListener
        extends StateMachineListenerAdapter<String, String> {

    @Override
    public void stateMachineError(StateMachine<String, String> stateMachine, Exception exception) {
        // do something with error
    }
}
```

Generic `ApplicationListener` checking `StateMachineEvent` would look like.

```
public class GenericApplicationEventListener
        implements ApplicationListener<StateMachineEvent> {

    @Override
    public void onApplicationEvent(StateMachineEvent event) {
        if (event instanceof OnStateMachineError) {
            // do something with error
        }
    }
}
```

It's also possible to define `ApplicationListener` directly to recognize only `StateMachineEvent` instances.

```java
public class ErrorApplicationEventListener
        implements ApplicationListener<OnStateMachineError> {

    @Override
    public void onApplicationEvent(OnStateMachineError event) {
        // do something with error
    }
}
```

22. Persisting State Machine

Traditionally an instance of a state machine is used as is within a running program. More dynamic behaviour is possible to achieve via dynamic builders and factories which allows state machine instantiation on-demand. Building an instance of a state machine is relatively heavy operation so if there is a need to i.e. handle arbitrary state change in a database using a state machine we need to find a better and faster way to do it.

Persist feature allows user to save a state of a state machine itself into an external repository and later reset a state machine based of serialized state. For example if you have a database table keeping orders it would be way too expensive to update order state via a state machine if a new instance would need to be build for every change. Persist feature allows you to reset a state machine state without instantiating a new state machine instance.

> **Note**
>
> There is one recipe Chapter 25, *Persist* and one sample Chapter 32, *Persist* which provides more info about persisting states.

While it is possible to build a custom persistence feature using a `StateMachineListener` it has one conceptual problem. When listener notifies a change of state, state change has already happened. If a custom persistent method within a listener fails to update serialized state in an external repository, state in a state machine and state in an external repository are then in inconsistent state.

State machine interceptor can be used instead of where attempt to save serialized state into an external storage is done during the a state change within a state machine. If this interceptor callback fails, state change attempt will be halted and instead of ending into an inconsistent state, user can then handle this error manually. Using the interceptors are discussed in Chapter 20, *State Machine Interceptor*.

23. Using Distributed States

Distributed state is probably one of a most compicated concepts of a Spring State Machine. What exactly is a distributed state? A state within a single state machine is naturally really simple to understand but when there is a need to introduce a shared distributed state through a state machines, things will get a little complicated.

> **Note**
>
> Distributed state functionality is still a preview feature and is not yet considered to be stable in this particular release. We expect this feature to mature towards the first official release.

For generic configuration support see section Section 8.9, “Configuring Common Settings” and actual usage example see sample Chapter 33, *Zookeeper*.

`Distributed State Machine` is implemented via a `DistributedStateMachine` class which simply wraps an actual instance of a `StateMachine`. `DistributedStateMachine` intercepts communication with a `StateMachine` instance and works with distributed state abstractions handled via interface `StateMachineEnsemble`. Depending on an actual implementation `StateMachinePersist` interface may also be used to serialize a `StateMachineContext` which contains enough information to reset a `StateMachine`.

While `Distributed State Machine` is implemented via an abstraction, only one implementation currently exists based on `Zookeeper`.

Here is a generic example of how `Zookeeper` based `Distributed State Machine` would be configured.

```
@Configuration
@EnableStateMachine
public class Config
        extends StateMachineConfigurerAdapter<String, String> {

    @Override
    public void configure(StateMachineConfigurationConfigurer<String, String> config)
            throws Exception {
        config
            .withDistributed()
                .ensemble(stateMachineEnsemble())
                .and()
            .withConfiguration()
                .autoStartup(true);
    }

    @Override
    public void configure(StateMachineStateConfigurer<String, String> states)
            throws Exception {
        // config states
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<String, String> transitions)
            throws Exception {
        // config transitions
    }

    @Bean
    public StateMachineEnsemble<String, String> stateMachineEnsemble()
            throws Exception {
        return new ZookeeperStateMachineEnsemble<String, String>(curatorClient(), "/zkpath");
    }

    @Bean
    public CuratorFramework curatorClient()
            throws Exception {
        CuratorFramework client = CuratorFrameworkFactory
                .builder()
                .defaultData(new byte[0])
                .connectString("localhost:2181").build();
        client.start();
        return client;
    }

}
```

Current technical documentation of a `Zookeeker` based distributed state machine can be found from an appendice Appendix C, *Distributed State Machine Technical Paper*.

23.1 ZookeeperStateMachineEnsemble

`ZookeeperStateMachineEnsemble` itself needs two mandatory settings, an instance of `curatorClient` and `basePath`. Client is a `CuratorFramework` and path is root of a tree in a `Zookeeper`.

Optionally it is possible to set `cleanState` which defaults to `TRUE` and will clear existing data if no members exists in an ensemble. Set this to `FALSE` if you want to preserve distributed state within application restarts.

Optionally it is possible to set a size of a `logSize` which defaults to `32` and is used to keep history of state changes. Value of this setting needs to be a power of two. `32` is generally good default value but if a particular state machine is left behind more than a size of a log it is put into error state and disconnected from an ensemble indicating it has lost its history to reconstruct fully synchronized status.

24. Testing Support

We have also added a set of utility classes to easy testing of a state machine instances. These are used in a framework itself but are also very useful for end users.

`StateMachineTestPlanBuilder` is used to build a `StateMachineTestPlan` which then have one method `test()` which runs a plan. `StateMachineTestPlanBuilder` contains a fluent builder api to add steps into a plan and during these steps you can send events and check various conditions like state changes, transitions and extended state variables.

Let’s take a simple `StateMachine` build using below example:

```
private StateMachine<String, String> buildMachine() throws Exception {
    StateMachineBuilder.Builder<String, String> builder = StateMachineBuilder.builder();

    builder.configureConfiguration()
        .withConfiguration()
            .taskExecutor(new SyncTaskExecutor())
            .autoStartup(true);

    builder.configureStates()
            .withStates()
                .initial("SI")
                .state("S1");

    builder.configureTransitions()
            .withExternal()
                .source("SI").target("S1")
                .event("E1");

    return builder.build();
}
```

In below test plan we have two steps, first we check that initial state `S1` is indeed set, secondly we send an event `E1` and expect one state change to happen and machine to end up into a state `S1`.

```
StateMachine<String, String> machine = buildMachine();
StateMachineTestPlan<String, String> plan =
        StateMachineTestPlanBuilder.<String, String>builder()
            .defaultAwaitTime(2)
            .stateMachine(machine)
            .step()
                .expectStates("SI")
                .and()
            .step()
                .sendEvent("E1")
                .expectStateChanged(1)
                .expectStates("S1")
                .and()
            .build();
plan.test();
```

These utilities are also used within a framework to test distributed state machine features and multiple machines can be added to a plan. If multiple machines are added then it is also possible to choose if event is sent to particular, random or all machines.

Part IV. Recipes

This chapter contains documentation for existing built-in state machine recipes.

What exactly is a recipe? As Spring Statemachine is always going to be a foundational framework meaning that its core will not have that much higher level functionality or dependencies outside of a Spring Framework. Correct usage of a state machine may be a little difficult time to time and there’s always some common use cases how state machine can be used. Recipe modules are meant to provide a higher level solutions to these common use cases and also provide examples beyond samples how framework can be used.

Note

Recipes are a great way to make external contributions this Spring Statemachine project. If you’re not ready to contribute to the framework core itself, a custom and common recipe is a great way to share functionality among other users.

25. Persist

Persist recipe is a simple utility which allows to use a single state machine instance to persist and update a state of an arbitrary item in a repository.

Recipe’s main class is `PersistStateMachineHandler` which assumes user to do three different things:

- An instance of a `StateMachine<String, String>` needs to be used with a `PersistStateMachineHandler`. States and Events are required to be type of Strings.
- `PersistStateChangeListener` need to be registered with handler order to react to persist request.
- Method `handleEventWithState` is used to orchestrate state changes.

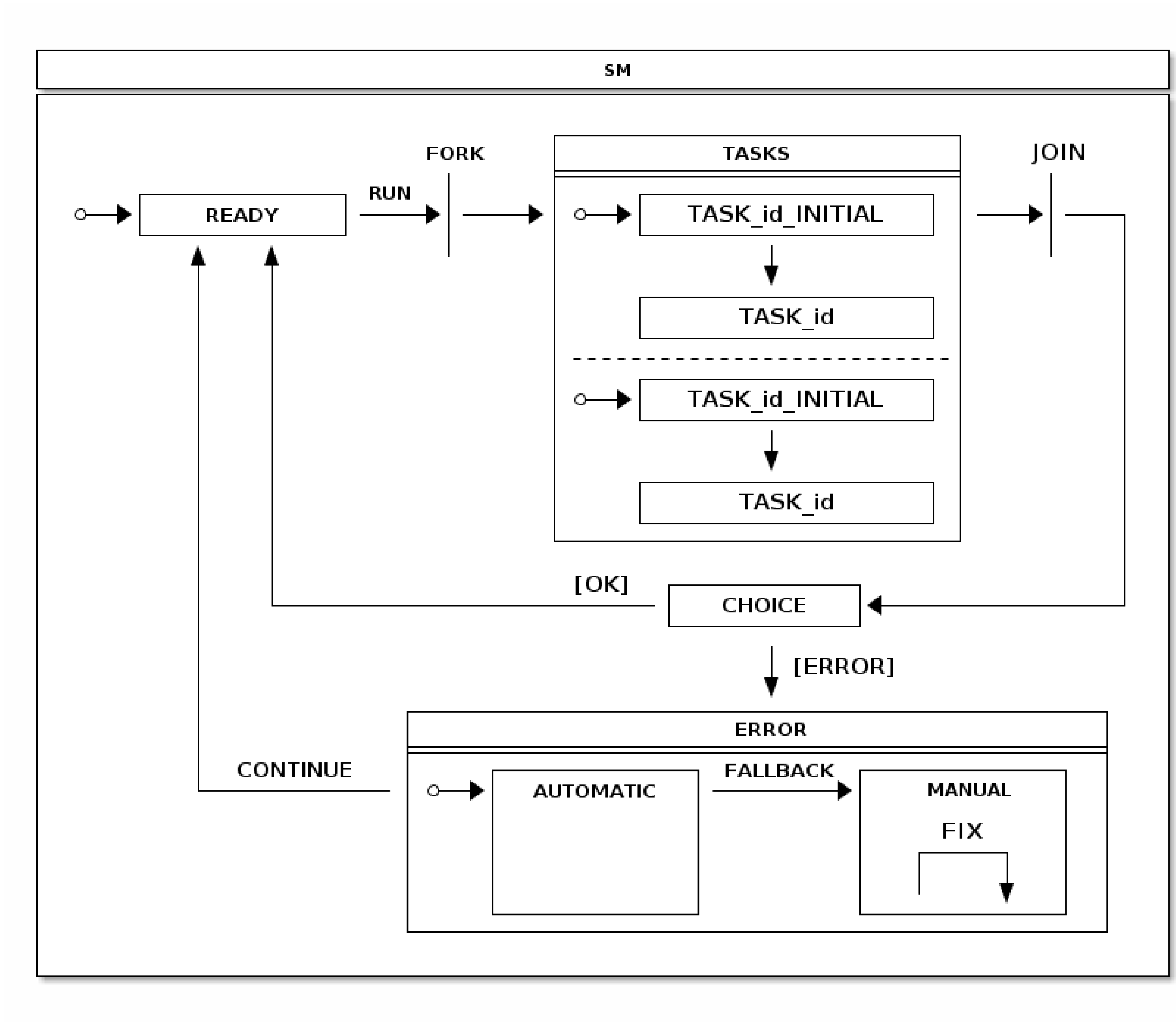
There is a sample demonstrating usage of this recipe at Chapter 32, *Persist*.

26. Tasks

Tasks recipe is a concept to execute DAG of `Runnable` instances using a state machine. This recipe has been developed from ideas introduced in sample Chapter 30, *Tasks*.

Generic concept of a state machine is shown below. In this state chart everything under TASKS just shows a generic concept of how a single task is executed. Because this recipe allows to register deep hierarcical DAG of tasks, meaning a real state chart would be deep nested collection of sub-states and regions, there's no need to be more presise.

For example if you have only two registered tasks, below state chart would be correct with `TASK_id` replaced with `TASK_1` and `TASK_2` if registered tasks ids are `1` and `2`.



Executing a `Runnable` may result an error and especially if a complex DAG of tasks is involved it is desirable that there is a way to handle tasks execution errors and then having a way to continue execution without executing already successfully executed tasks. Addition to this it would be nice if some execution errors can be handled automatically and as a last fallback, if error can't be handled automatically, state machine is put into a state where user can handle errors manually.

`TasksHandler` contains a builder method to configure handler instance and follows a simple builder patter. This builder can be used to register `Runnable` tasks, `TasksListener` instances, define `StateMachinePersist` hook, and setup custom `TaskExecutor` instance.

Now lets take a simple `Runnable` just doing a simple sleep as shown below. This is a base of all examples in this chapter.

```
private Runnable sleepRunnable() {
    return new Runnable() {

        @Override
        public void run() {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
            }
        }
    };
}
```

To execute multiple `sleepRunnable` tasks just register tasks and execute `runTasks()` method from `TasksHandler`.

```
TasksHandler handler = TasksHandler.builder()
        .task("1", sleepRunnable())
        .task("2", sleepRunnable())
        .task("3", sleepRunnable())
        .build();

handler.runTasks();
```

Order to listen what is happening with a task execution an instance of a `TasksListener` can be registered with a `TasksHandler`. Recipe provides an adapter `TasksListenerAdapter` if you don't want to implement a full interface. Listener provides a various hooks to listen tasks execution events.

```
private class MyTasksListener extends TasksListenerAdapter {

    @Override
    public void onTasksStarted() {
    }

    @Override
    public void onTasksContinue() {
    }

    @Override
    public void onTaskPreExecute(Object id) {
    }

    @Override
    public void onTaskPostExecute(Object id) {
    }

    @Override
    public void onTaskFailed(Object id, Exception exception) {
    }

    @Override
    public void onTaskSuccess(Object id) {
    }

    @Override
    public void onTasksSuccess() {
    }

    @Override
    public void onTasksError() {
    }

    @Override
    public void onTasksAutomaticFix(TasksHandler handler, StateContext<String, String> context) {
    }
}
```

Listeners can be either registered via a builder or directly with a `TasksHandler` as shown above.

```
MyTasksListener listener1 = new MyTasksListener();
MyTasksListener listener2 = new MyTasksListener();

TasksHandler handler = TasksHandler.builder()
        .task("1", sleepRunnable())
        .task("2", sleepRunnable())
        .task("3", sleepRunnable())
        .listener(listener1)
        .build();

handler.addTasksListener(listener2);
handler.removeTasksListener(listener2);

handler.runTasks();
```

Above sample show how to create a deep nested DAG of tasks. Every task needs to have an unique identifier and optionally as task can be defined to be a sub-task. Effectively this will create a DAG of tasks.

```
TasksHandler handler = TasksHandler.builder()
        .task("1", sleepRunnable())
        .task("1", "12", sleepRunnable())
        .task("1", "13", sleepRunnable())
        .task("2", sleepRunnable())
        .task("2", "22", sleepRunnable())
        .task("2", "23", sleepRunnable())
        .task("3", sleepRunnable())
        .task("3", "32", sleepRunnable())
        .task("3", "33", sleepRunnable())
        .build();

handler.runTasks();
```

When error happens and a state machine running these tasks goes into a `ERROR` state, user can call handler methods `fixCurrentProblems` to reset current state of tasks kept in a state machine extended state variables. Handler method `continueFromError` can then be used to instruct state machine to transition from `ERROR` state back to `READY` state where tasks can be executed again.

```
TasksHandler handler = TasksHandler.builder()
        .task("1", sleepRunnable())
        .task("2", sleepRunnable())
        .task("3", sleepRunnable())
        .build();

        handler.runTasks();
        handler.fixCurrentProblems();
        handler.continueFromError();
```

Part V. State Machine Examples

This part of the reference documentation explains the use of state machines together with a sample code and a uml state charts. We do few shortcuts when representing relationship between a state chart, SSM configuration and what an application does with a state machine. For complete examples go and study the samples repository.

Samples are build directly from a main source distribution during a normal build cycle. Samples in this chapter are:

Chapter 27, *Turnstile* Turnstile.

Chapter 28, *Showcase* Showcase.

Chapter 29, *CD Player* CD Player.

Chapter 30, *Tasks* Tasks.

Chapter 31, *Washer* Washer.

Chapter 32, *Persist* Persist.

Chapter 33, *Zookeeper* Zookeeper.

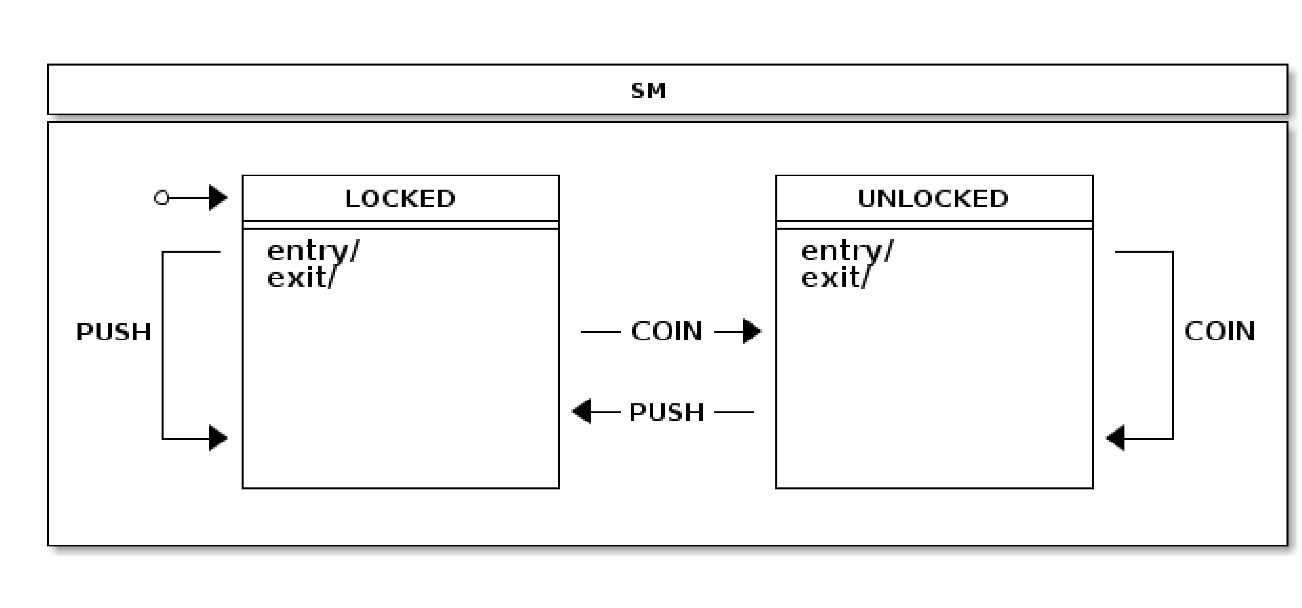
Chapter 34, *Web* Web.

```
./gradlew clean build -x test
```

Every sample is located in its own directory under `spring-statemachine-samples`. Samples are based on spring-boot and spring-shell and you will find usual boot fat jars under every sample projects `build/libs` directory.

27. Turnstile

Turnstile is a simple device which gives you an access if payment is made and is a very simple to model using a state machine. In its simplest form there are only two states, `LOCKED` and `UNLOCKED`. Two events, `COIN` and `PUSH` can happen if you try to go through it or you make a payment.



States.

```
public enum States {
    LOCKED, UNLOCKED
}
```

Events.

```
public enum Events {
    COIN, PUSH
}
```

Configuration.

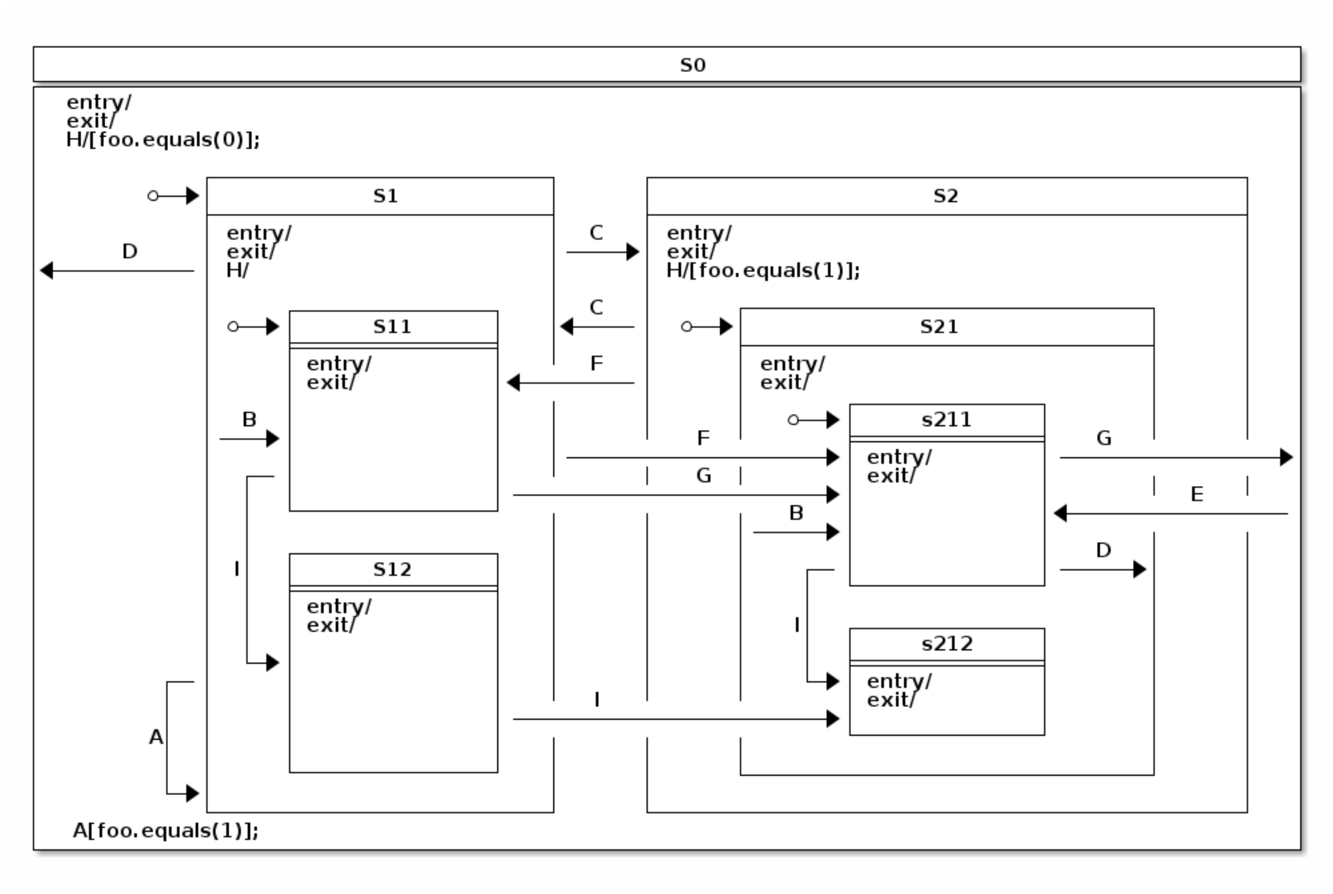
```
@Configuration
@EnableStateMachine
static class StateMachineConfig
        extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.LOCKED)
                .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.LOCKED)
                .target(States.UNLOCKED)
                .event(Events.COIN)
                .and()
            .withExternal()
                .source(States.UNLOCKED)
                .target(States.LOCKED)
                .event(Events.PUSH);
    }

}
```

You can see how this sample state machine interacts with event by running `turnstile` sample.

```
$ java -jar spring-statemachine-samples-turnstile-1.0.0.BUILD-SNAPSHOT.jar

sm>sm print
+----------------------------------------------------------------+
|                              SM                                |
+----------------------------------------------------------------+
|                                                                |
|         +----------------+          +----------------+         |
|     *-->|     LOCKED     |          |    UNLOCKED    |         |
|         +----------------+          +----------------+         |
|     +---| entry/         |          | entry/         |---+     |
|     |   | exit/          |          | exit/          |   |     |
|     |   |                |          |                |   |     |
| PUSH|   |                |---COIN-->|                |   |COIN |
|     |   |                |          |                |   |     |
|     |   |                |          |                |   |     |
|     |   |                |<--PUSH---|                |   |     |
|     +-->|                |          |                |<--+     |
|         |                |          |                |         |
|         +----------------+          +----------------+         |
|                                                                |
+----------------------------------------------------------------+

sm>sm start
State changed to LOCKED
State machine started

sm>sm event COIN
State changed to UNLOCKED
Event COIN send

sm>sm event PUSH
State changed to LOCKED
Event PUSH send
```

28. Showcase

Showcase is a complex state machine showing all possible transition topologies up to four levels of state nesting.



States.

```
public enum States {
    S0, S1, S11, S12, S2, S21, S211, S212
}
```

Events.

```
public enum Events {
    A, B, C, D, E, F, G, H, I
}
```

Configuration - states.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.S0, fooAction())
            .state(States.S0)
            .and()
            .withStates()
                .parent(States.S0)
                .initial(States.S1)
                .state(States.S1)
                .and()
                .withStates()
                    .parent(States.S1)
                    .initial(States.S11)
                    .state(States.S11)
                    .state(States.S12)
                    .and()
            .withStates()
                .parent(States.S0)
                .state(States.S2)
                .and()
                .withStates()
                    .parent(States.S2)
                    .initial(States.S21)
                    .state(States.S21)
                    .and()
                    .withStates()
                        .parent(States.S21)
                        .initial(States.S211)
                        .state(States.S211)
                        .state(States.S212);
}
```

Configuration - transitions.

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.S1).target(States.S1).event(Events.A)
            .guard(foo1Guard())
            .and()
        .withExternal()
            .source(States.S1).target(States.S11).event(Events.B)
            .and()
        .withExternal()
            .source(States.S21).target(States.S211).event(Events.B)
            .and()
        .withExternal()
            .source(States.S1).target(States.S2).event(Events.C)
            .and()
        .withExternal()
            .source(States.S2).target(States.S1).event(Events.C)
            .and()
        .withExternal()
            .source(States.S1).target(States.S0).event(Events.D)
            .and()
        .withExternal()
            .source(States.S211).target(States.S21).event(Events.D)
            .and()
        .withExternal()
            .source(States.S0).target(States.S211).event(Events.E)
            .and()
        .withExternal()
            .source(States.S1).target(States.S211).event(Events.F)
            .and()
        .withExternal()
            .source(States.S2).target(States.S11).event(Events.F)
            .and()
        .withExternal()
            .source(States.S11).target(States.S211).event(Events.G)
            .and()
        .withExternal()
            .source(States.S211).target(States.S0).event(Events.G)
            .and()
        .withInternal()
            .source(States.S0).event(Events.H)
            .guard(foo0Guard())
            .action(fooAction())
            .and()
        .withInternal()
            .source(States.S2).event(Events.H)
            .guard(foo1Guard())
            .action(fooAction())
            .and()
        .withInternal()
            .source(States.S1).event(Events.H)
            .and()
        .withExternal()
            .source(States.S11).target(States.S12).event(Events.I)
            .and()
        .withExternal()
            .source(States.S211).target(States.S212).event(Events.I)
            .and()
        .withExternal()
            .source(States.S12).target(States.S212).event(Events.I);

}
```

Configuration - actions and guard.

```
@Bean
public FooGuard foo0Guard() {
    return new FooGuard(0);
}

@Bean
public FooGuard foo1Guard() {
    return new FooGuard(1);
}

@Bean
public FooAction fooAction() {
    return new FooAction();
}
```

Action.

```
private static class FooAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        Map<Object, Object> variables = context.getExtendedState().getVariables();
        Integer foo = context.getExtendedState().get("foo", Integer.class);
        if (foo == null) {
            log.info("Init foo to 0");
            variables.put("foo", 0);
        } else if (foo == 0) {
            log.info("Switch foo to 1");
            variables.put("foo", 1);
        } else if (foo == 1) {
            log.info("Switch foo to 0");
            variables.put("foo", 0);
        }
    }
}
```

Guard.

```
private static class FooGuard implements Guard<States, Events> {

    private final int match;

    public FooGuard(int match) {
        this.match = match;
    }

    @Override
    public boolean evaluate(StateContext<States, Events> context) {
        Object foo = context.getExtendedState().getVariables().get("foo");
        return !(foo == null || !foo.equals(match));
    }
}
```

Let’s go through what this state machine do when it’s executed and we send various event to it.

```
sm>sm start
Init foo to 0
Entry state S0
Entry state S1
Entry state S11
State machine started

sm>sm event A
Event A send

sm>sm event C
Exit state S11
Exit state S1
Entry state S2
Entry state S21
Entry state S211
Event C send

sm>sm event H
Switch foo to 1
Internal transition source=S0
Event H send

sm>sm event C
Exit state S211
Exit state S21
Exit state S2
Entry state S1
Entry state S11
Event C send

sm>sm event A
Exit state S11
Exit state S1
Entry state S1
Entry state S11
Event A send
```

What happens in above sample:

- State machine is started which takes it to its initial state *S11* via superstates *S1* and *S0*. Also extended state variable `foo` is init to `0`.
- We try to execute self transition in state *S1* with event *A* but nothing happens because transition is guarded by variable `foo` to be `1`.
- We send event *C* which takes us to other state machine where initial state *S211* and its superstates are entered. In there we can use event *H* which does a simple internal transition to flip variable `foo`. Then we simply go back using event *C*.
- Event *A* is sent again and now *S1* does a self transition because guard evaluates true.

Let’s take closer look of how hierarchical states and their event handling works with a below example.

```
sm>sm variables
No variables

sm>sm start
Init foo to 0
Entry state S0
Entry state S1
Entry state S11
State machine started

sm>sm variables
foo=0

sm>sm event H
Internal transition source=S1
Event H send

sm>sm variables
foo=0

sm>sm event C
Exit state S11
Exit state S1
Entry state S2
Entry state S21
Entry state S211
Event C send

sm>sm variables
foo=0

sm>sm event H
Switch foo to 1
Internal transition source=S0
Event H send

sm>sm variables
foo=1

sm>sm event H
Switch foo to 0
Internal transition source=S2
Event H send

sm>sm variables
foo=0
```

What happens in above sample:

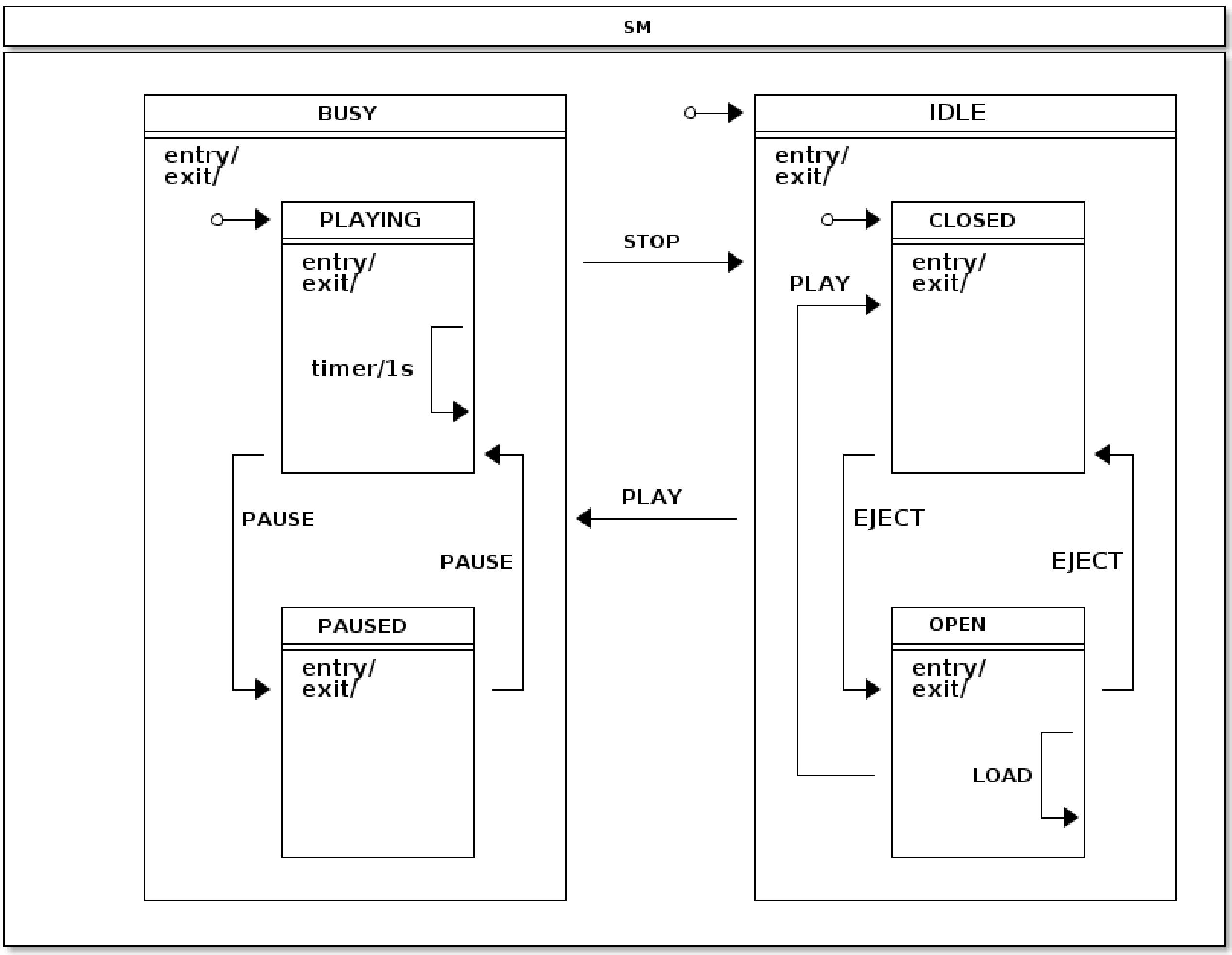
- We print extended state variables in various stages.
- With event *H* we end up executing internal transition which is logged with source state.
- It’s also worth to pay attention to how event *H* is handled in different states *S0*, *S1* and *S2*. This is a good example of how hierarchical states and their event handling works. If state *S2* is unable to handle event *H* due to guard condition, its parent is checked next. This guarantees that while on state *S2*, `foo` flag is always flipped around. However in state *S1* event *H* always match to its dummy transition without guard or action, not never happens.

29. CD Player

CD Player is a sample which resembles better use case of most of use have used in a real world. CD Player itself is a really simple entity where user can open a deck, insert or change a disk, then drive player functionality by pressing various buttons like *eject*, *play*, *stop*, *pause*, *rewind* and *backward*.

How many of us have really given a thought of what it will take to make a code for a CD Player which interacts with a hardware. Yes, concept of a player is overly simple but if you look behind the scenes things actually get a bit convoluted.

You’ve probably noticed that if your deck is open and you press play, deck will close and a song will start to play if CD was inserted in a first place. In a sense when deck is open you first need to close it and then try to start playing if cd is actually inserted. Hopefully you have now realised that a simple CD Player is not anymore so simple. Sure you can wrap all this with a simple class with few boolean variables and probably few nested if/else clauses, that will do the job, but what about if you need to make all this behaviour much more complex, do you really want to keep adding more flags and if/else clauses.



Let’s go through how this sample and its state machine is designed and how those two interacts with each other. Below three config sections are used withing a *EnumStateMachineConfigurerAdapter*.

```java
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.IDLE)
            .state(States.IDLE)
            .and()
            .withStates()
                .parent(States.IDLE)
                .initial(States.CLOSED)
                .state(States.CLOSED, closedEntryAction(), null)
                .state(States.OPEN)
                .and()
        .withStates()
            .state(States.BUSY)
            .and()
            .withStates()
                .parent(States.BUSY)
                .initial(States.PLAYING)
                .state(States.PLAYING)
                .state(States.PAUSED);

}
```

```java
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.CLOSED).target(States.OPEN).event(Events.EJECT)
            .and()
        .withExternal()
            .source(States.OPEN).target(States.CLOSED).event(Events.EJECT)
            .and()
        .withExternal()
            .source(States.OPEN).target(States.CLOSED).event(Events.PLAY)
            .and()
        .withExternal()
            .source(States.PLAYING).target(States.PAUSED).event(Events.PAUSE)
            .and()
        .withInternal()
            .source(States.PLAYING)
            .action(playingAction())
            .timer(1000)
            .and()
        .withInternal()
            .source(States.PLAYING).event(Events.BACK)
            .action(trackAction())
            .and()
        .withInternal()
            .source(States.PLAYING).event(Events.FORWARD)
            .action(trackAction())
            .and()
        .withExternal()
            .source(States.PAUSED).target(States.PLAYING).event(Events.PAUSE)
            .and()
        .withExternal()
            .source(States.BUSY).target(States.IDLE).event(Events.STOP)
            .and()
        .withExternal()
            .source(States.IDLE).target(States.BUSY).event(Events.PLAY)
            .action(playAction())
            .guard(playGuard())
            .and()
        .withInternal()
            .source(States.OPEN).event(Events.LOAD).action(loadAction());
}
```

```
@Bean
public ClosedEntryAction closedEntryAction() {
    return new ClosedEntryAction();
}

@Bean
public LoadAction loadAction() {
    return new LoadAction();
}

@Bean
public TrackAction trackAction() {
    return new TrackAction();
}

@Bean
public PlayAction playAction() {
    return new PlayAction();
}

@Bean
public PlayingAction playingAction() {
    return new PlayingAction();
}

@Bean
public PlayGuard playGuard() {
    return new PlayGuard();
}
```

What we did in above configuration:

- We used EnumStateMachineConfigurerAdapter to configure states and transitions.
- States *CLOSED* and *OPEN* are defined as substates of *IDLE*, states *PLAYING* and *PAUSED* are defined as substates of *BUSY*.
- With state *CLOSED* we added entry action as bean *closedEntryAction*.
- With transition we mostly mapped events to expected state transitions like *EJECT* closing and opening a deck, *PLAY*, *STOP* and *PAUSE* doing their natural transitions. Few words to mention what we did for other transitions.
  - With source state *PLAYING* we added a timer trigger which is needed to automatically track elapsed time within a playing track and to have facility to make a decision when to switch to next track.
  - With event *PLAY* if source state is *IDLE* and target state is *BUSY* we defined action *playAction* and guard *playGuard*.
  - With event *LOAD* and state *OPEN* we defined internal transition with action *loadAction* which will insert cd disc into extended state variables.
  - *PLAYING* state defined three internal transitions where one is triggered by a timer executing a *playingAction* which updates extended state variables. Other two transitions are with *trackAction* with different events, *BACK* and *FORWARD* respectively which handles when user wants to go back or forward in tracks.

This machine only have six states which are introduced as an enum.

```
public enum States {
    // super state of PLAYING and PAUSED
    BUSY,
    PLAYING,
    PAUSED,
    // super state of CLOSED and OPEN
    IDLE,
    CLOSED,
    OPEN
}
```

Events represent, in a sense in this example, what buttons user would press and if user loads a cd disc into a deck.

```
public enum Events {
    PLAY, STOP, PAUSE, EJECT, LOAD, FORWARD, BACK
}
```

Beans *cdPlayer* and *library* are just used with a sample to drive the application.

```
@Bean
public CdPlayer cdPlayer() {
    return new CdPlayer();
}

@Bean
public Library library() {
    return Library.buildSampleLibrary();
}
```

We can define extended state variable key as simple enums.

```
public enum Variables {
    CD, TRACK, ELAPSEDTIME
}

public enum Headers {
    TRACKSHIFT
}
```

We wanted to make this samply type safe so we’re defining our own annotation *@StatesOnTransition* which have a mandatory meta annotation *@OnTransition*.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@OnTransition
public @interface StatesOnTransition {

    States[] source() default {};

    States[] target() default {};

}
```

ClosedEntryAction is a entry action for state *CLOSED* to simply send and *PLAY* event to a statemachine if cd disc is present.

```
public static class ClosedEntryAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        if (context.getTransition() != null
                && context.getEvent() == Events.PLAY
                && context.getTransition().getTarget().getId() == States.CLOSED
                && context.getExtendedState().getVariables().get(Variables.CD) != null) {
            context.getStateMachine().sendEvent(Events.PLAY);
        }
    }
}
```

LoadAction is simply updating extended state variable if event headers contained information about a cd disc to load.

```
public static class LoadAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        Object cd = context.getMessageHeader(Variables.CD);
        context.getExtendedState().getVariables().put(Variables.CD, cd);
    }
}
```

PlayAction is simply resetting player elapsed time which is kept as an extended state variable.

```
public static class PlayAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        context.getExtendedState().getVariables().put(Variables.ELAPSEDTIME, 0l);
        context.getExtendedState().getVariables().put(Variables.TRACK, 0);
    }
}
```

PlayGuard is used to guard transition from *IDLE* to *BUSY* with event *PLAY* if extended state variable *CD* doesn’t indicate that cd disc has been loaded.

```
public static class PlayGuard implements Guard<States, Events> {

    @Override
    public boolean evaluate(StateContext<States, Events> context) {
        ExtendedState extendedState = context.getExtendedState();
        return extendedState.getVariables().get(Variables.CD) != null;
    }
}
```

PlayingAction is updating extended state variable *ELAPSEDTIME* which cd player itself can read and update lcd status. Action also handles track shift if user is going back or forward in tracks.

```
public static class PlayingAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        Map<Object, Object> variables = context.getExtendedState().getVariables();
        Object elapsed = variables.get(Variables.ELAPSEDTIME);
        Object cd = variables.get(Variables.CD);
        Object track = variables.get(Variables.TRACK);
        if (elapsed instanceof Long) {
            long e = ((Long)elapsed) + 1000l;
            if (e > ((Cd) cd).getTracks()[((Integer) track)].getLength()*1000) {
                context.getStateMachine().sendEvent(MessageBuilder
                        .withPayload(Events.FORWARD)
                        .setHeader(Headers.TRACKSHIFT.toString(), 1).build());
            } else {
                variables.put(Variables.ELAPSEDTIME, e);
            }
        }
    }
}
```

TrackAction handles track shift action if user is going back or forward in tracks. If it is a last track of a cd, playing is stopped and *STOP* event sent to a state machine.

```
public static class TrackAction implements Action<States, Events> {

    @Override
    public void execute(StateContext<States, Events> context) {
        Map<Object, Object> variables = context.getExtendedState().getVariables();
        Object trackshift = context.getMessageHeader(Headers.TRACKSHIFT.toString());
        Object track = variables.get(Variables.TRACK);
        Object cd = variables.get(Variables.CD);
        if (trackshift instanceof Integer && track instanceof Integer && cd instanceof Cd) {
            int next = ((Integer)track) + ((Integer)trackshift);
            if (next >= 0 &&  ((Cd)cd).getTracks().length > next) {
                variables.put(Variables.ELAPSEDTIME, 0l);
                variables.put(Variables.TRACK, next);
            } else if (((Cd)cd).getTracks().length <= next) {
                context.getStateMachine().sendEvent(Events.STOP);
            }
        }
    }
}
```

One other important aspect of a state machines is that they have their own responsibilities mostly around handling states and all application level logic should be kept outside. This means that application needs to have a ways to interact with a state machine and below sample is how cdplayer does it order to update lcd status. Also pay attention that we annotated *CdPlayer* with *@WithStateMachine* which instructs state machine to find methods from your pojo which are then called with various transitions.

```
@OnTransition(target = "BUSY")
public void busy(ExtendedState extendedState) {
    Object cd = extendedState.getVariables().get(Variables.CD);
    if (cd != null) {
        cdStatus = ((Cd)cd).getName();
    }
}
```

In above example we use *@OnTransition* annotation to hook a callback when transition happens with a target state *BUSY*.

```
@StatesOnTransition(target = {States.CLOSED, States.IDLE})
public void closed(ExtendedState extendedState) {
    Object cd = extendedState.getVariables().get(Variables.CD);
    if (cd != null) {
        cdStatus = ((Cd)cd).getName();
    } else {
        cdStatus = "No CD";
    }
    trackStatus = "";
}
```

@OnTransition we used above can only be used with strings which are matched from enums. *@StatesOnTransition* is then something what user can create into his own application to get a type safe annotation where a real enums can be used.

Let's see an example how this state machine actually works.

```
sm>sm start
Entry state IDLE
Entry state CLOSED
State machine started

sm>cd lcd
No CD

sm>cd library
0: Greatest Hits
  0: Bohemian Rhapsody  05:56
  1: Another One Bites the Dust  03:36
1: Greatest Hits II
  0: A Kind of Magic  04:22
  1: Under Pressure  04:08

sm>cd eject
Exit state CLOSED
Entry state OPEN

sm>cd load 0
Loading cd Greatest Hits

sm>cd play
Exit state OPEN
Entry state CLOSED
Exit state CLOSED
Exit state IDLE
Entry state BUSY
Entry state PLAYING

sm>cd lcd
Greatest Hits Bohemian Rhapsody 00:03

sm>cd forward

sm>cd lcd
Greatest Hits Another One Bites the Dust 00:04

sm>cd stop
Exit state PLAYING
Exit state BUSY
Entry state IDLE
Entry state CLOSED

sm>cd lcd
Greatest Hits
```

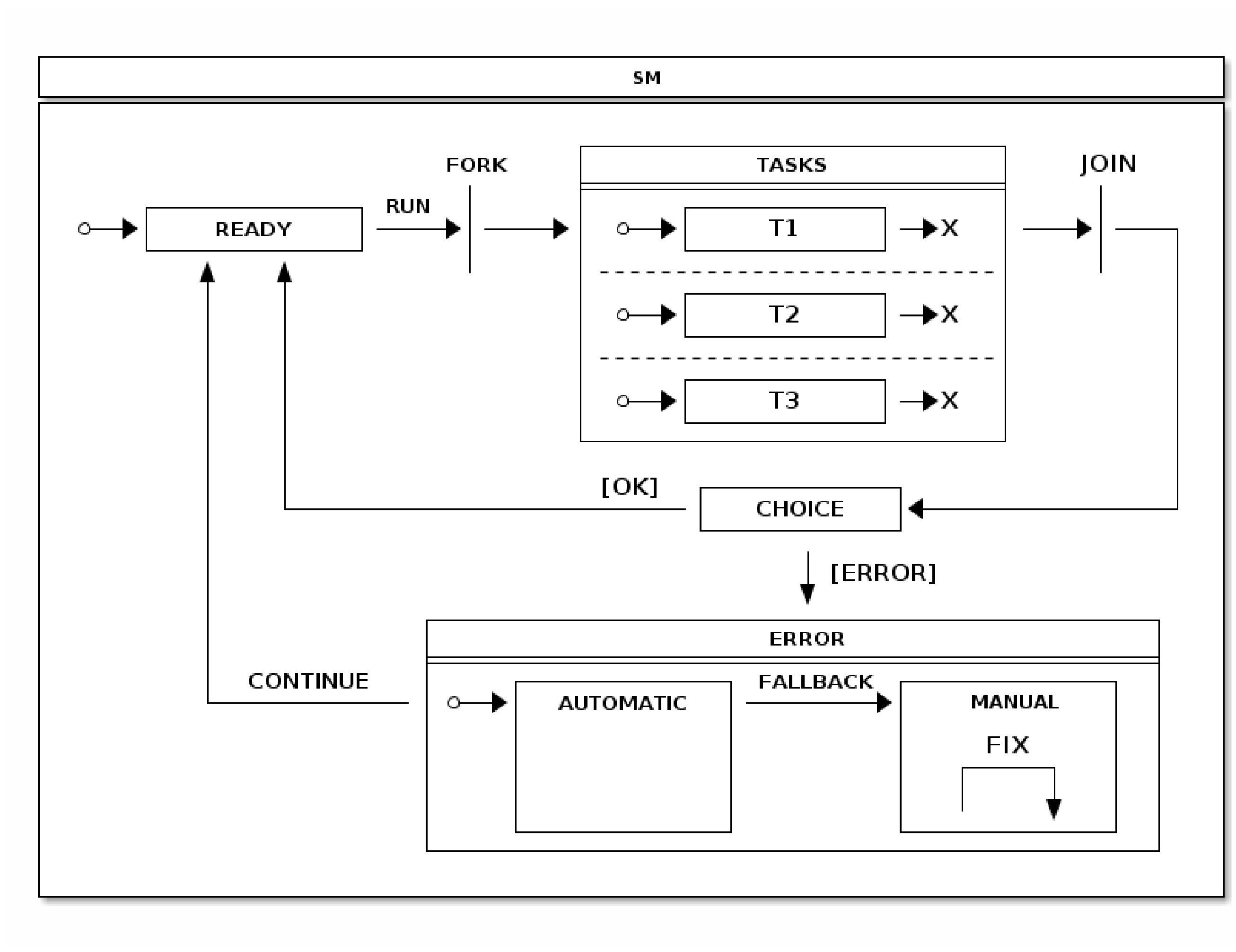
What happened in above run:

- State machine is started which causes machine to get initialized.

- CD Player lcd screen status is printed.
- CD Library is printed.
- CD Player deck is opened.
- CD with index 0 is loaded into a deck.
- Play is causing deck to get closed and immediate playing because cd was inserted.
- We print lcd status and request next track.
- We stop playing.

30. Tasks

Tasks is a sample demonstrating a parallel task handling within a regions and additionally adds an error handling to either automatically or manually fixing task problems before continuing back to a state where tasks can be run again.



On a high level what happens in this state machine is:

- We’re always trying to get into READY state so that we can use event RUN to execute tasks.
- TASKS state which is composed with 3 independent regions has been put in a middle of FORK and JOIN states which will cause regions to go into its initial states and to be joined by end states.
- From JOIN state we go automatically into a CHOICE state which checks existence of error flags in extended state variables. Tasks can set these flags and it gives CHOICE state a possibility to go into ERROR state where errors can be handled either automatically or manually.
- AUTOMATIC state in ERROR can try to automatically fix error and goes back to READY if it succeed to do so. If error is something what can’t be handled automatically, user intervention is needed and machine is put into MANUAL state via FALLBACK event.

States.

```
public enum States {
    READY,
    FORK, JOIN, CHOICE,
    TASKS, T1, T1E, T2, T2E, T3, T3E,
    ERROR, AUTOMATIC, MANUAL
}
```

Events.

```
public enum Events {
    RUN, FALLBACK, CONTINUE, FIX;
}
```

Configuration - states.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.READY)
            .fork(States.FORK)
            .state(States.TASKS)
            .join(States.JOIN)
            .choice(States.CHOICE)
            .state(States.ERROR)
            .and()
            .withStates()
                .parent(States.TASKS)
                .initial(States.T1)
                .end(States.T1E)
                .and()
            .withStates()
                .parent(States.TASKS)
                .initial(States.T2)
                .end(States.T2E)
                .and()
            .withStates()
                .parent(States.TASKS)
                .initial(States.T3)
                .end(States.T3E)
                .and()
            .withStates()
                .parent(States.ERROR)
                .initial(States.AUTOMATIC)
                .state(States.AUTOMATIC, automaticAction(), null)
                .state(States.MANUAL);
}
```

Configuration - transitions.

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.READY).target(States.FORK)
            .event(Events.RUN)
            .and()
        .withFork()
            .source(States.FORK).target(States.TASKS)
            .and()
        .withExternal()
            .source(States.T1).target(States.T1E)
            .and()
        .withExternal()
            .source(States.T2).target(States.T2E)
            .and()
        .withExternal()
            .source(States.T3).target(States.T3E)
            .and()
        .withJoin()
            .source(States.TASKS).target(States.JOIN)
            .and()
        .withExternal()
            .source(States.JOIN).target(States.CHOICE)
            .and()
        .withChoice()
            .source(States.CHOICE)
            .first(States.ERROR, tasksChoiceGuard())
            .last(States.READY)
            .and()
        .withExternal()
            .source(States.ERROR).target(States.READY)
            .event(Events.CONTINUE)
            .and()
        .withExternal()
            .source(States.AUTOMATIC).target(States.MANUAL)
            .event(Events.FALLBACK)
            .and()
        .withInternal()
            .source(States.MANUAL)
            .action(fixAction())
            .event(Events.FIX);
}
```

Guard below is guarding choice entry into a ERROR state and needs to return TRUE if error has happened. For this guard simply checks that all extended state variables(T1, T2 and T3) are TRUE.

```
@Bean
public Guard<States, Events> tasksChoiceGuard() {
    return new Guard<States, Events>() {

        @Override
        public boolean evaluate(StateContext<States, Events> context) {
            Map<Object, Object> variables = context.getExtendedState().getVariables();
            return !(ObjectUtils.nullSafeEquals(variables.get("T1"), true)
                    && ObjectUtils.nullSafeEquals(variables.get("T2"), true)
                    && ObjectUtils.nullSafeEquals(variables.get("T3"), true));
        }
    };
}
```

Actions below will simply send event to a state machine to request next step which would be either fallback or continue back to ready.

```
@Bean
public Action<States, Events> automaticAction() {
    return new Action<States, Events>() {

        @Override
        public void execute(StateContext<States, Events> context) {
            Map<Object, Object> variables = context.getExtendedState().getVariables();
            if (ObjectUtils.nullSafeEquals(variables.get("T1"), true)
                    && ObjectUtils.nullSafeEquals(variables.get("T2"), true)
                    && ObjectUtils.nullSafeEquals(variables.get("T3"), true)) {
                context.getStateMachine().sendEvent(Events.CONTINUE);
            } else {
                context.getStateMachine().sendEvent(Events.FALLBACK);
            }
        }
    };
}

@Bean
public Action<States, Events> fixAction() {
    return new Action<States, Events>() {

        @Override
        public void execute(StateContext<States, Events> context) {
            Map<Object, Object> variables = context.getExtendedState().getVariables();
            variables.put("T1", true);
            variables.put("T2", true);
            variables.put("T3", true);
            context.getStateMachine().sendEvent(Events.CONTINUE);
        }
    };
}
```

Currently default region execution is synchronous but it can be changed to asynchronous by changing `TaskExecutor`. Task will simulate work by sleeping 2 seconds so you'll able to see how actions in regions are executed parallel.

```
@Bean(name = StateMachineSystemConstants.TASK_EXECUTOR_BEAN_NAME)
public TaskExecutor taskExecutor() {
    ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();
    taskExecutor.setCorePoolSize(5);
    return taskExecutor;
}
```

Let's see an examples how this state machine actually works.

```
sm>sm start
State machine started
Entry state READY

sm>tasks run
Entry state TASKS
run task on T3
run task on T2
run task on T1
run task on T2 done
run task on T1 done
run task on T3 done
Entry state T2
Entry state T3
Entry state T1
Entry state T1E
Entry state T2E
Entry state T3E
Exit state TASKS
Entry state JOIN
Exit state JOIN
Entry state READY
```

In above we can execute tasks multiple times.

```
sm>tasks list
Tasks {T1=true, T3=true, T2=true}

sm>tasks fail T1

sm>tasks list
Tasks {T1=false, T3=true, T2=true}

sm>tasks run
Entry state TASKS
run task on T1
run task on T3
run task on T2
run task on T1 done
run task on T3 done
run task on T2 done
Entry state T1
Entry state T3
Entry state T2
Entry state T1E
Entry state T2E
Entry state T3E
Exit state TASKS
Entry state JOIN
Exit state JOIN
Entry state ERROR
Entry state AUTOMATIC
Exit state AUTOMATIC
Exit state ERROR
Entry state READY
```

In above, if we simulate failure for task T1, it is fixed automatically.

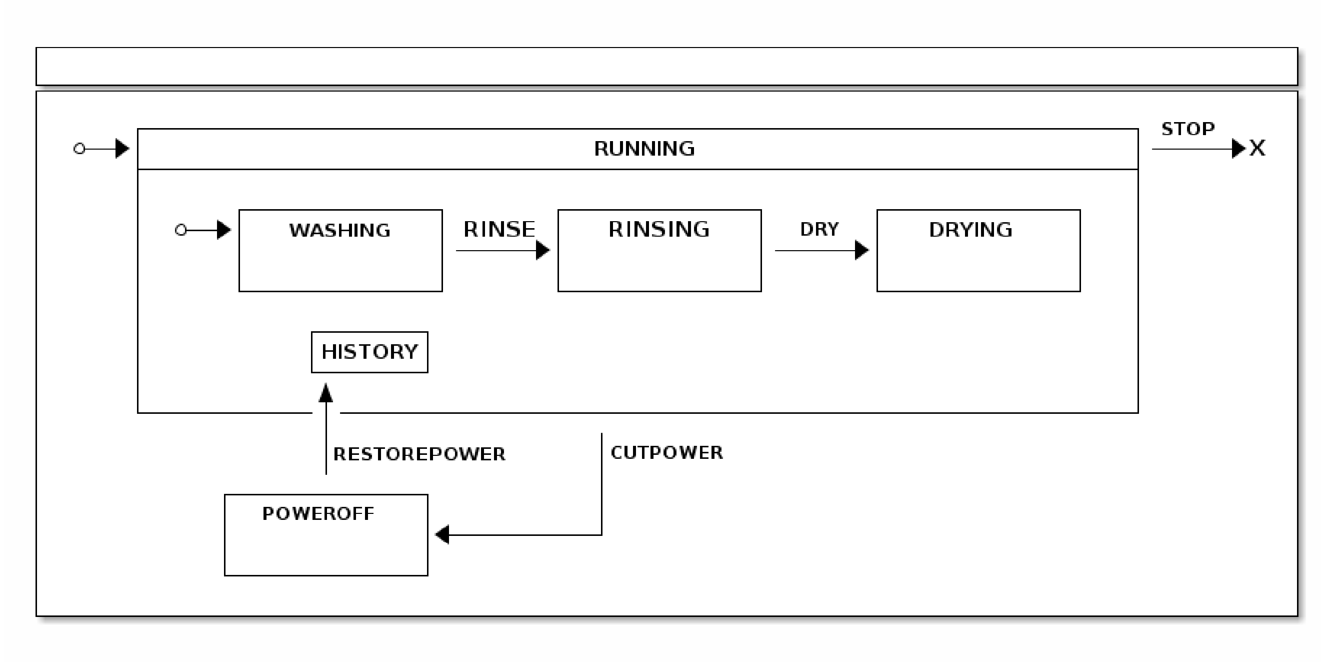
```
sm>tasks list
Tasks {T1=true, T3=true, T2=true}

sm>tasks fail T2

sm>tasks run
Entry state TASKS
run task on T2
run task on T1
run task on T3
run task on T2 done
run task on T1 done
run task on T3 done
Entry state T2
Entry state T1
Entry state T3
Entry state T1E
Entry state T2E
Entry state T3E
Exit state TASKS
Entry state JOIN
Exit state JOIN
Entry state ERROR
Entry state AUTOMATIC
Exit state AUTOMATIC
Entry state MANUAL

sm>tasks fix
Exit state MANUAL
Exit state ERROR
Entry state READY
```

In above if we simulate failure for either task T2 or T3, state machine goes to MANUAL state where problem needs to be fixed manually before we’re able to go back to READY state.

31. Washer

Washer is a sample demonstrating a use of a history state to recover a running state configuration with a simulated power off situation.

Anyone ever used a washing machine knows that if you can somehow pause the program it will continue from a same state when lid is closed. This kind of behaviour can be implemented in a state machine by using a history pseudo state.



States.

```
public enum States {
    RUNNING, HISTORY, END,
    WASHING, RINSING, DRYING,
    POWEROFF
}
```

Events.

```
public enum Events {
    RINSE, DRY, STOP,
    RESTOREPOWER, CUTPOWER
}
```

Configuration - states.

```
@Override
public void configure(StateMachineStateConfigurer<States, Events> states)
        throws Exception {
    states
        .withStates()
            .initial(States.RUNNING)
            .state(States.POWEROFF)
            .end(States.END)
            .and()
            .withStates()
                .parent(States.RUNNING)
                .initial(States.WASHING)
                .state(States.RINSING)
                .state(States.DRYING)
                .history(States.HISTORY, History.SHALLOW);
}
```

Configuration - transitions.

```
@Override
public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    transitions
        .withExternal()
            .source(States.WASHING).target(States.RINSING)
            .event(Events.RINSE)
            .and()
        .withExternal()
            .source(States.RINSING).target(States.DRYING)
            .event(Events.DRY)
            .and()
        .withExternal()
            .source(States.RUNNING).target(States.POWEROFF)
            .event(Events.CUTPOWER)
            .and()
        .withExternal()
            .source(States.POWEROFF).target(States.HISTORY)
            .event(Events.RESTOREPOWER)
            .and()
        .withExternal()
            .source(States.RUNNING).target(States.END)
            .event(Events.STOP);
}
```

Let’s see an example how this state machine actually works.

```
sm>sm start
Entry state RUNNING
Entry state WASHING
State machine started

sm>sm event RINSE
Exit state WASHING
Entry state RINSING
Event RINSE send

sm>sm event DRY
Exit state RINSING
Entry state DRYING
Event DRY send

sm>sm event CUTPOWER
Exit state DRYING
Exit state RUNNING
Entry state POWEROFF
Event CUTPOWER send

sm>sm event RESTOREPOWER
Exit state POWEROFF
Entry state RUNNING
Entry state WASHING
Entry state DRYING
Event RESTOREPOWER send
```

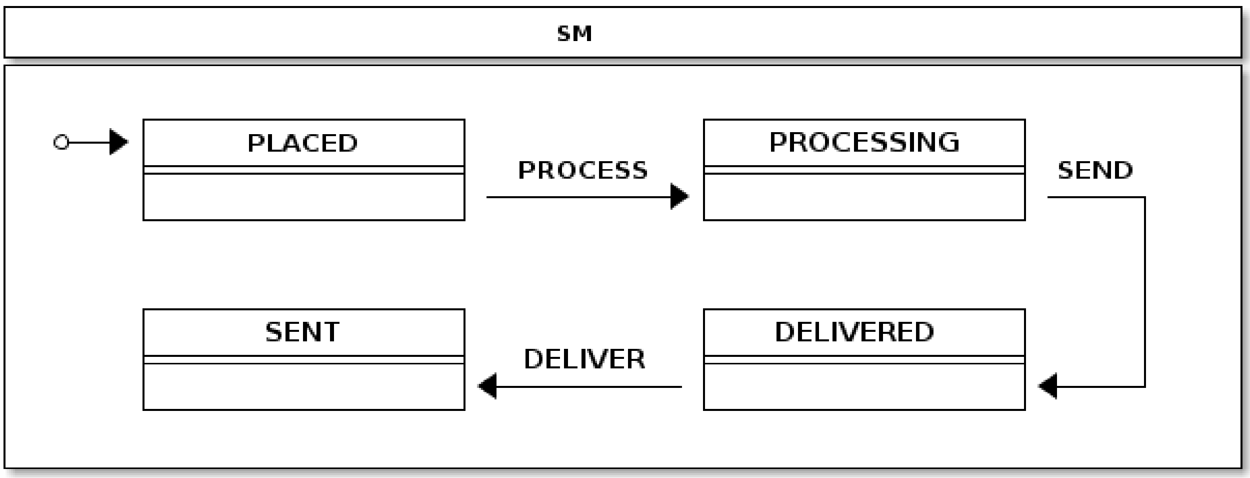
What happened in above run:

- State machine is started which causes machine to get initialized.
- We go to RINSING state.
- We go to DRYING state.
- We cut power and go to POWEROFF state.
- State is restored via HISTORY state which takes state machine back to its previous known state.

32. Persist

Persist is a sample using recipe Chapter 25, *Persist* to demonstrate how a database entry update logic can be controlled by a state machine.

The state machine logic and configuration is shown above:



StateMachine Config.

```
@Configuration
@EnableStateMachine
static class StateMachineConfig
        extends StateMachineConfigurerAdapter<String, String> {

    @Override
    public void configure(StateMachineStateConfigurer<String, String> states)
            throws Exception {
        states
            .withStates()
                .initial("PLACED")
                .state("PROCESSING")
                .state("SENT")
                .state("DELIVERED");
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<String, String> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source("PLACED").target("PROCESSING")
                .event("PROCESS")
                .and()
            .withExternal()
                .source("PROCESSING").target("SENT")
                .event("SEND")
                .and()
            .withExternal()
                .source("SENT").target("DELIVERED")
                .event("DELIVER");
    }

}
```

`PersistStateMachineHandler` can be created using a below config:

Handler Config.

```
@Configuration
static class PersistHandlerConfig {

    @Autowired
    private StateMachine<String, String> stateMachine;

    @Bean
    public Persist persist() {
        return new Persist(persistStateMachineHandler());
    }

    @Bean
    public PersistStateMachineHandler persistStateMachineHandler() {
        return new PersistStateMachineHandler(stateMachine);
    }

}
```

Order class used with this sample is shown below:

Order Class.

```
public static class Order {
    int id;
    String state;

    public Order(int id, String state) {
        this.id = id;
        this.state = state;
    }

    @Override
    public String toString() {
        return "Order [id=" + id + ", state=" + state + "]";
    }

}
```

Now let’s see how this example works.

```
sm>persist db
Order [id=1, state=PLACED]
Order [id=2, state=PROCESSING]
Order [id=3, state=SENT]
Order [id=4, state=DELIVERED]

sm>persist process 1
Exit state PLACED
Entry state PROCESSING

sm>persist db
Order [id=2, state=PROCESSING]
Order [id=3, state=SENT]
Order [id=4, state=DELIVERED]
Order [id=1, state=PROCESSING]

sm>persist deliver 3
Exit state SENT
Entry state DELIVERED

sm>persist db
Order [id=2, state=PROCESSING]
Order [id=4, state=DELIVERED]
Order [id=1, state=PROCESSING]
Order [id=3, state=DELIVERED]
```

What happened in above run:

- We listed rows from an existing embedded database which is already populated with sample data.
- We request to update order `1` into `PROCESSING` state.
- We list db entries again and see that state has been changed from `PLACED` into a `PROCESSING`.
- We do update for order 3 to update state from `SENT` into `DELIVERED`.

Note

If you're wondering where is the database because there are literally no signs of it in a sample code. Sample is based on Spring Boot and because necessary classes are in a classpath, embedded `HSQL` instance is created automatically.

Spring Boot will even create an instance of `JdbcTemplate` which you can just autowire like how it's done in `Persist.java`.

```
@Autowired
private JdbcTemplate jdbcTemplate;
```

Finally we need to handle state changes:

```
public void change(int order, String event) {
    Order o = jdbcTemplate.queryForObject("select id, state from orders where id = ?", new Object[]
 { order },
            new RowMapper<Order>() {
                public Order mapRow(ResultSet rs, int rowNum) throws SQLException {
                    return new Order(rs.getInt("id"), rs.getString("state"));
                }
            });
    handler.handleEventWithState(MessageBuilder.withPayload(event).setHeader("order", order).build(),
 o.state);
}
```

And use a `PersistStateChangeListener` to update database:

```
private class LocalPersistStateChangeListener implements PersistStateChangeListener {

    @Override
    public void onPersist(State<String, String> state, Message<String> message,
            Transition<String, String> transition, StateMachine<String, String> stateMachine) {
        if (message != null && message.getHeaders().containsKey("order")) {
            Integer order = message.getHeaders().get("order", Integer.class);
            jdbcTemplate.update("update orders set state = ? where id = ?", state.getId(), order);
        }
    }
}
```

33. Zookeeper

Zookeeper is a distributed version from sample Chapter 27, *Turnstile*.

Note

This sample needs and external `Zookeeper` instance accessible from `localhost` with default port and settings.

Configuration of this sample is almost same as `turnstile` sample. We only add configuration for distributed state machine where we configure `StateMachineEnsemble`.

```
@Override
public void configure(StateMachineConfigurationConfigurer<String, String> config) throws Exception {
    config
        .withDistributed()
            .ensemble(stateMachineEnsemble());
}
```

Actual `StateMachineEnsemble` needs to be created as bean together with `CuratorFramework` client.

```
@Bean
public StateMachineEnsemble<String, String> stateMachineEnsemble() throws Exception {
    return new ZookeeperStateMachineEnsemble<String, String>(curatorClient(), "/foo");
}

@Bean
public CuratorFramework curatorClient() throws Exception {
    CuratorFramework client = CuratorFrameworkFactory.builder().defaultData(new byte[0])
            .retryPolicy(new ExponentialBackoffRetry(1000, 3))
            .connectString("localhost:2181").build();
    client.start();
    return client;
}
```

Let's go through a simple example where two different shell instances are started with command `java -jar spring-statemachine-samples-zookeeper-1.0.0.BUILD-SNAPSHOT.jar`.

First open first shell instance(do not start second instance yet). When state machine is started it will end up into its initial state `LOCKED`. Then send event `COIN` to transit into `UNLOCKED` state.

Shell1.

```
sm>sm start
Entry state LOCKED
State machine started

sm>sm event COIN
Exit state LOCKED
Entry state UNLOCKED
Event COIN send

sm>sm state
UNLOCKED
```

Open second shell instance and start a state machine. You should see that distributed state `UNLOCKED` is entered instead of default initial state `LOCKED`.

Shell2.

```
sm>sm start
State machine started

sm>sm state
UNLOCKED
```

Then from either of a shells(we use second instance here) send event `PUSH` to transit from `UNLOCKED` into `LOCKED` state.

Shell2.

```
sm>sm event PUSH
Exit state UNLOCKED
Entry state LOCKED
Event PUSH send
```

In other shell you should see state getting changed automatically based on distributed state kept in Zookeeper.

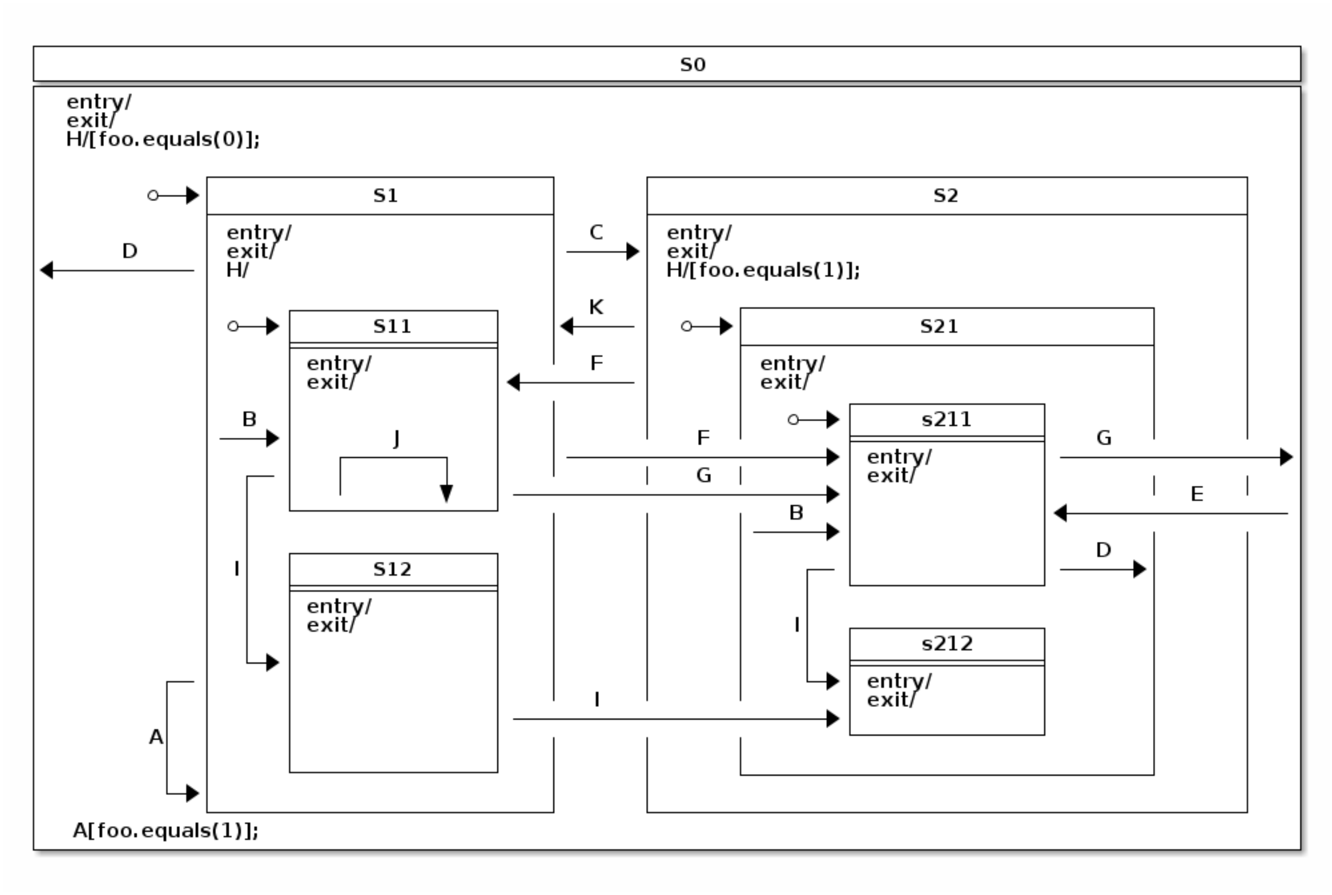
Shell1.

```
sm>Exit state UNLOCKED
Entry state LOCKED
```

34. Web

Web is a distributed state machine example using a zookeeper to handle distributed state. This example is meant to be run on a multiple browser sessions against a multiple different hosts.

This sample is using a modified state machine structure from a Chapter 28, *Showcase* to work with a distributed state machine. The state machine logic is shown above:



Note

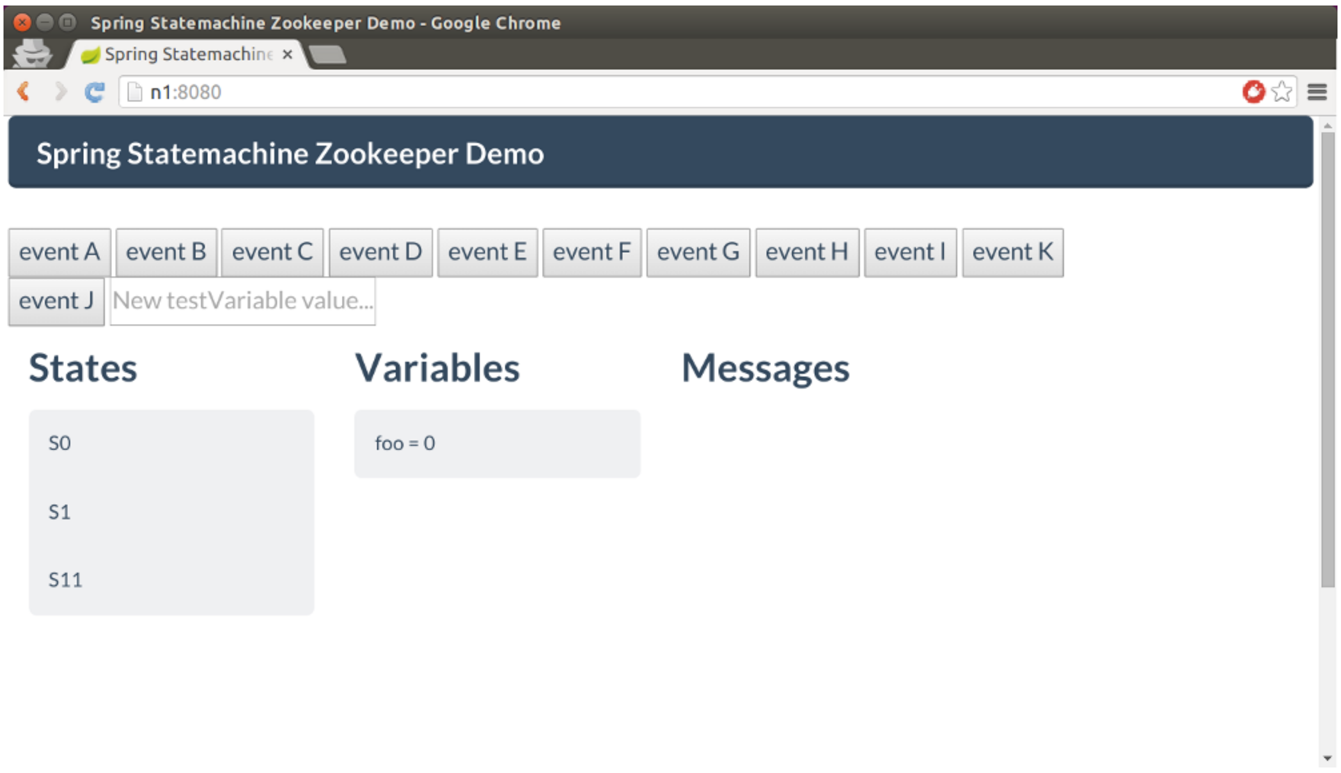
Due to nature of this sample an instance of a `Zookeeper` is expected to be available from a localhost for every individual sample instance.

Let's go through a simple example where three different sample instances are started with command `java -jar spring-statemachine-samples-web-1.0.0.BUILD-SNAPSHOT.jar`. If you are running different instances on a same host you need to distinguish used port by adding `--server.port=<myport>` to the command. Otherwise default port for each host will be `8080`.

In this sample run we have three hosts, `n1`, `n2` and `n3` which all have a local zookeeper instance running and a state machine sample running on a port `8080`.

```
@n1:~# java -jar spring-statemachine-samples-web-1.0.0.BUILD-SNAPSHOT.jar
@n2:~# java -jar spring-statemachine-samples-web-1.0.0.BUILD-SNAPSHOT.jar
@n3:~# java -jar spring-statemachine-samples-web-1.0.0.BUILD-SNAPSHOT.jar
```

When all instances are running you should see all showing similar information via a browser where states are `S0`, `S1` and `S11`, and extended state variable `foo=0`. Main state is `S11`.



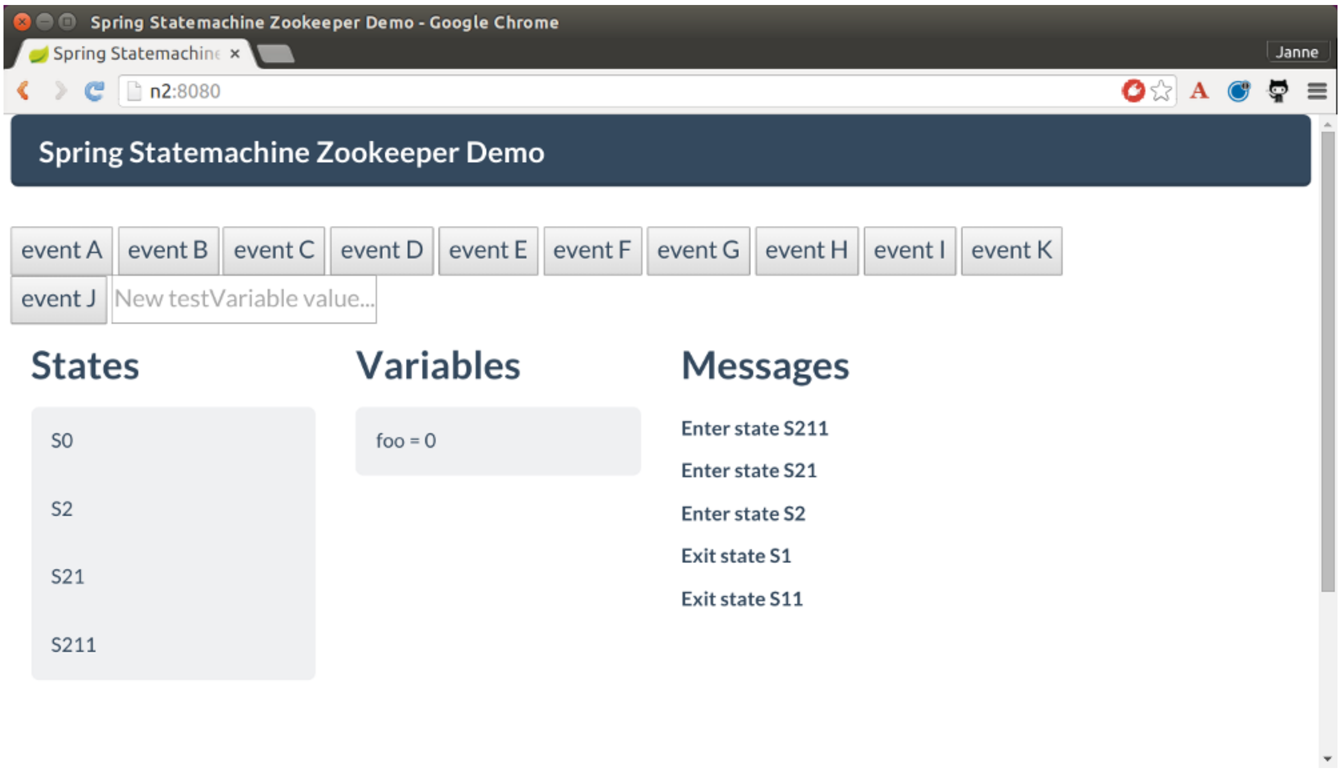
S0

S1

S11

foo = 0

When you press button `Event C` in any of a browser window, distributed state is changed to `S211` which is the target state denoted by transition associated with an event `C`.



S0

S2

S21

S211

foo = 0

Enter state S211

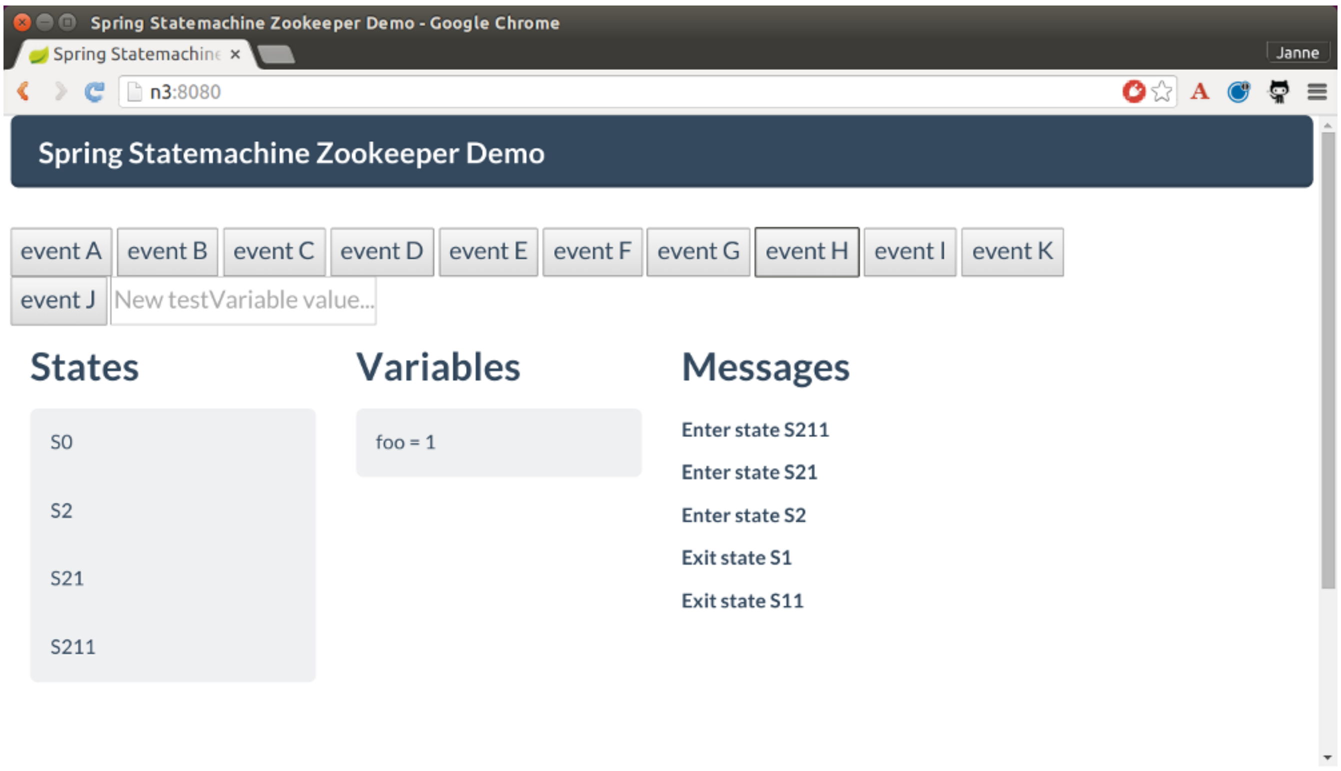
Enter state S21

Enter state S2

Exit state S1

Exit state S11

Then let’s press button `Event H` and what is supposed to happen is that internal transition is executed on all state machines changing extended state variable `foo` from value `0` to `1`. This change is first done on a state machine receiving the event and then propagated to other state machines. You should only see variable `foo` to change from `0` to `1`.


Spring Statemachine Zookeeper Demo - Google Chrome
Spring Statemachine
Janne
n3:8080
Spring Statemachine Zookeeper Demo
event A
event B
event C
event D
event E
event F
event G
event H
event I
event K
event J
New testVariable value...

States

S0

S2

S21

S211

Variables

foo = 1

Messages

Enter state S211

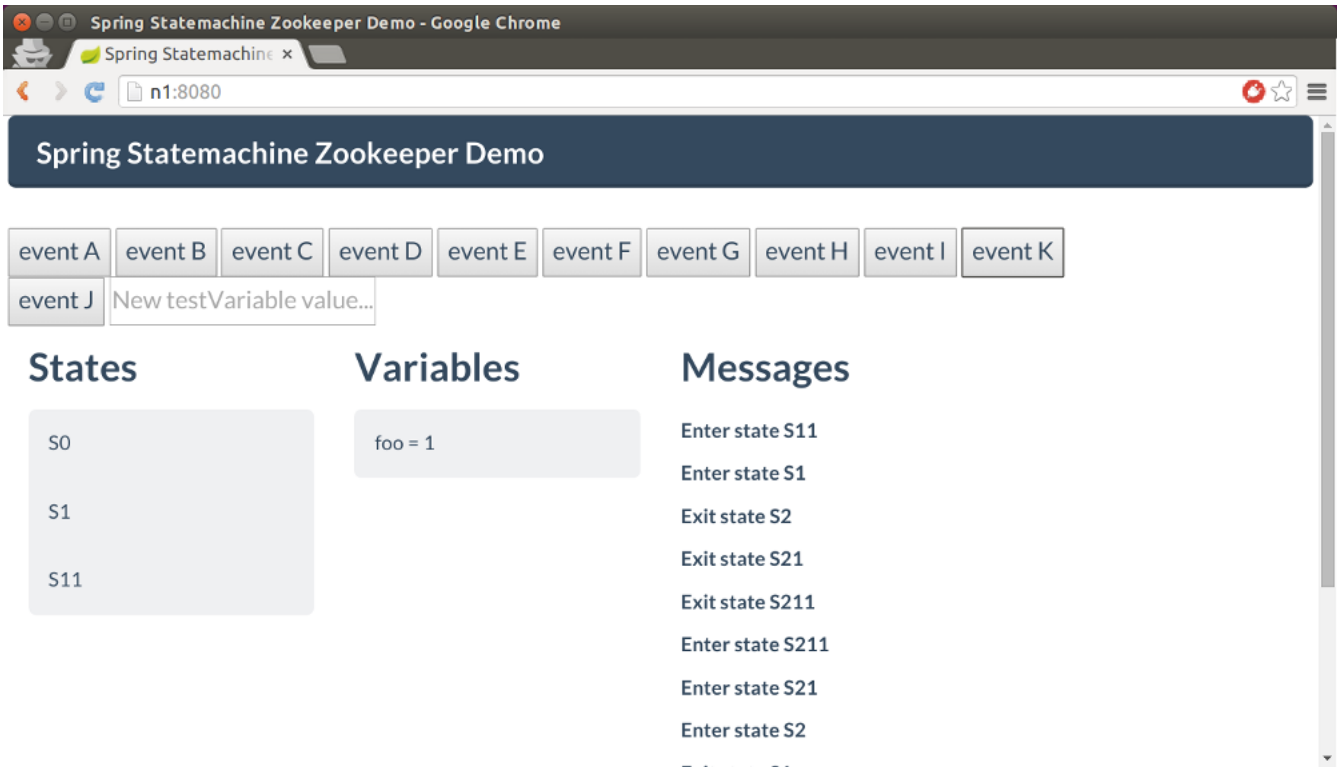
Enter state S21

Enter state S2

Exit state S1

Exit state S11

Last we simply send an event `Event K` which is supposed to take state machine state back to state `S11` and you should see this happening in all browser sessions.


Spring Statemachine Zookeeper Demo - Google Chrome
Spring Statemachine
n1:8080
Spring Statemachine Zookeeper Demo
event A
event B
event C
event D
event E
event F
event G
event H
event I
event K
event J
New testVariable value...

States

S0

S1

S11

Variables

foo = 1

Messages

Enter state S11

Enter state S1

Exit state S2

Exit state S21

Exit state S211

Enter state S211

Enter state S21

Enter state S2

Part VI. FAQ

This chapter tries to give solutions to question user is most likely to ask.

35. State Changes

I want to transit to next state automatically.

There are few choices a state machine developer can choose.

- Implement an action and send appropriate event into a state machine which triggers a transition into a proper target state.
- Define deferred event within a state and before sending an event send an event which will be deferred and thus causing next appropriate state transition when it is more convenient to handle that event.
- Implement a triggerless transition which will automatically cause state transition into a next state when state has entry and its actions has been completed.

36. Extented State

How I can initialise variables on state machine start.

Important concept in a state machine is that nothing really happens unless there is a trigger which is causing a state transition which then can fire actions. However, having said that, Spring Statemachine always have an initial transition when state machine is started. With this initial transition user can execute a simple action which within a *StateContext* can do whatever it likes with an extended state variables.

Part VII. Appendices

Appendix A. Support Content

This appendix provides generic information about used classes and material in this reference documentation.

A.1 Classes Used in This Document

```
public enum States {
    SI,S1,S2,S3,S4,SF
}
```

```
public enum States2 {
    S1,S2,S3,S4,S5,
    S2I,S21,S22,S2F,
    S3I,S31,S32,S3F
}
```

```
public enum States3 {
    S1,S2,SH,
    S2I,S21,S22,S2F
}
```

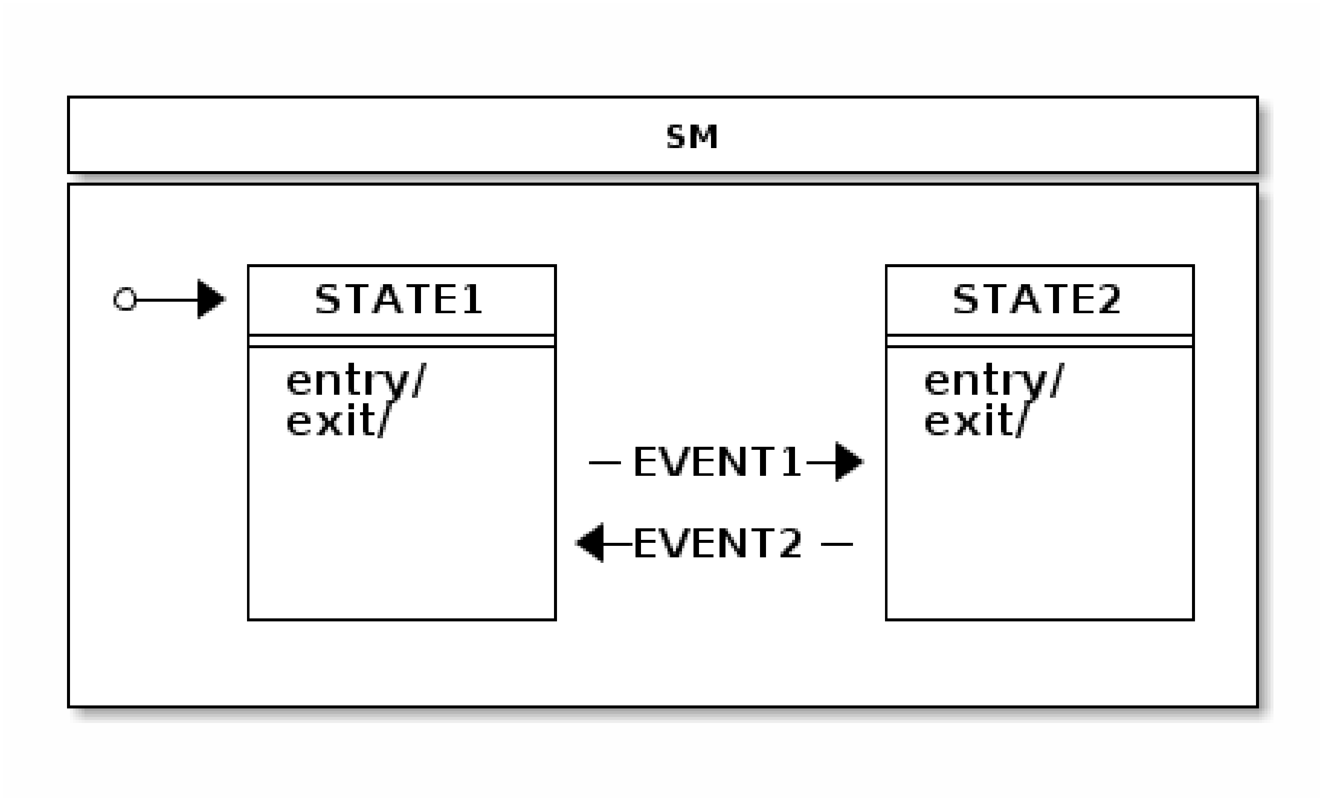
```
public enum Events {
    E1,E2,E3,E4,EF
}
```

Appendix B. State Machine Concepts

This appendix provides generic information about state machines.

B.1 Quick Example

Assuming we have states *STATE1*, *STATE2* and events *EVENT1*, *EVENT2*, logic of state machine can be defined as shown in below quick example.



```
public enum States {
    STATE1, STATE2
}

public enum Events {
    EVENT1, EVENT2
}
```

```
@Configuration
@EnableStateMachine
public class Config1 extends EnumStateMachineConfigurerAdapter<States, Events> {

    @Override
    public void configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states
            .withStates()
                .initial(States.STATE1)
                .states(EnumSet.allOf(States.class));
    }

    @Override
    public void configure(StateMachineTransitionConfigurer<States, Events> transitions)
            throws Exception {
        transitions
            .withExternal()
                .source(States.STATE1).target(States.STATE2)
                .event(Events.EVENT1)
                .and()
            .withExternal()
                .source(States.STATE2).target(States.STATE1)
                .event(Events.EVENT2);
    }
}
```

```
@WithStateMachine
public class MyBean {

    @OnTransition(target = "STATE1")
    void toState1() {
    }

    @OnTransition(target = "STATE2")
    void toState2() {
    }
}
```

```
public class MyApp {

    @Autowired
    StateMachine<States, Events> stateMachine;

    void doSignals() {
        stateMachine.sendEvent(Events.EVENT1);
        stateMachine.sendEvent(Events.EVENT2);
    }
}
```

B.2 Glossary

State Machine
: Main entity driving a collection of states together with regions, transitions and events.

State
: A state models a situation during which some invariant condition holds. State is the main entity of a state machine where state changes are driven by an events.

Extended State
: An extended state is a special set of variables kept in a state machine to reduce number of needed states.

Transition
A transition is a relationship between a source state and a target state. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type.

Event
An entity which is send to a state machine which then drives a various state changes.

Initial State
A special state in which the state machine starts. Initial state is always bound to a particular state machine or a region. A state machine with a multiple regions may have a multiple initial states.

End State
Also called as a final state is a special kind of state signifying that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed.

History State
A pseudo state which allows a state machine to remember its last active state. Two types of history state exists, *shallow* which only remember top level state and *deep* which remembers active states in a sub-machines.

Choice State
A pseudo state which allows to make a transition choice based of i.e. event headers or extended state variables.

Fork State
A pseudo state which gives a controlled entry into a regions.

Join State
A pseudo state which gives a controlled exit from a regions.

Region
A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.

Guard
Is a boolean expression evaluated dynamically based on the value of extended state variables and event parameters. Guard conditions affect the behavior of a state machine by enabling actions or transitions only when they evaluate to TRUE and disabling them when they evaluate to FALSE.

Action
A action is a behaviour executed during the triggering of the transition.

B.3 A State Machines Crash Course

This appendix provides generic crash course to a state machine concepts.

States

A state is a model which a state machine can be in. It is always easier to describe state as a real world example rather than trying to abstract concepts with a generic documentation. For example let’s take a simple example of a keyboard most of us are using every single day. If you have a full keyboard which has normal keys on a left side and the numeric keypad on a right side you may have noticed that the numeric keypad may be in a two different states depending whether numlock is activated or not. If it is not active then typing will result navigation using arrows, etc. If numpad is active then typing will result numbers to be used. Essentially numpad part of a keyboard can be in two different states.

To relate state concept to programming it means that instead of using flags, nested if/else/break clauses or other impractical logic you simply rely on state, state variables or other interaction with a state machine.

Pseudo States

PseudoState is a special type of state which usually introduces more higher level logic into a state machine by either giving a state a special meaning like initial state. State machine can then internally react to these states by doing various actions available in UML state machine concepts.

Initial

Initial pseudostate state is always needed for every single state machine whether you have a simple one level state machine or more complex state machine composed with submachines or regions. Initial state simple defines where state machine should go when it starts and without it state machine is ill-formed.

End

Terminate pseudostate which is also called as end state will indicate that a particular state machine has reached its final state. Effectively this mean that a state machine will no longer process any events and will not transit to any other state. However in a case of submachines are regions, state machine is able to restart from its terminal state.

Choice

Choice pseudostate is used to choose a dynamic conditional branch of a transition from this state. Dynamic condition is evaluated by guards so that at least one and at most one branch is selected. Usually a simple if/elseif/else structure is used to make sure that at least one branch is selected. Otherwise state machine might end up in a deadlock and configuration would be ill-formed.

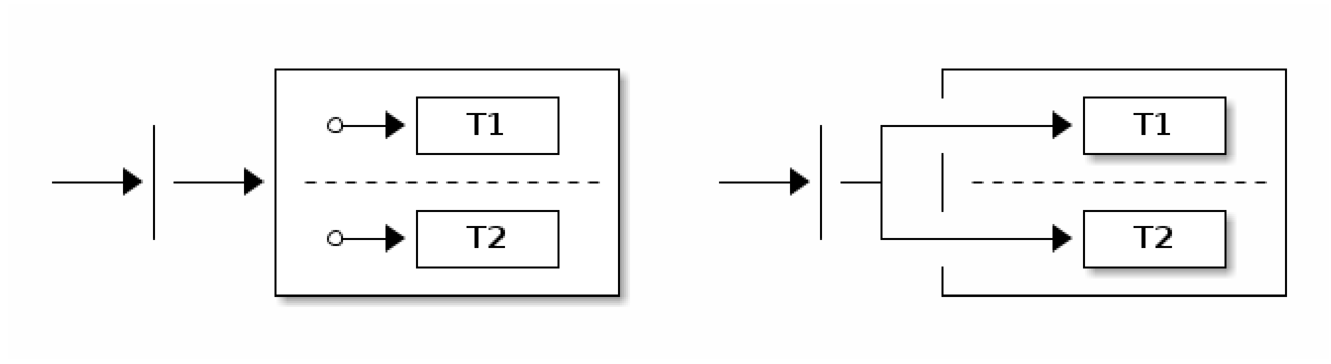
History

History pseudostate can be used to remember a last active state configuration. After state machine has been exited, history state can be used to restore previous knows configuration. There are two types of history states available, *SHALLOW* only remember active state of a state machine itself while *DEEP* also remembers nested states.

History state could be implemented externally by listening state machine events but this would soon make logic very difficult to work with, especially if state machine contains complex nested structures. Letting state machine itself to handle recording of history states makes things much simpler. What is left for user to do is simply do a transition into a history state and state machine will hand the needed logic to go back to its last known recorded state.

Fork

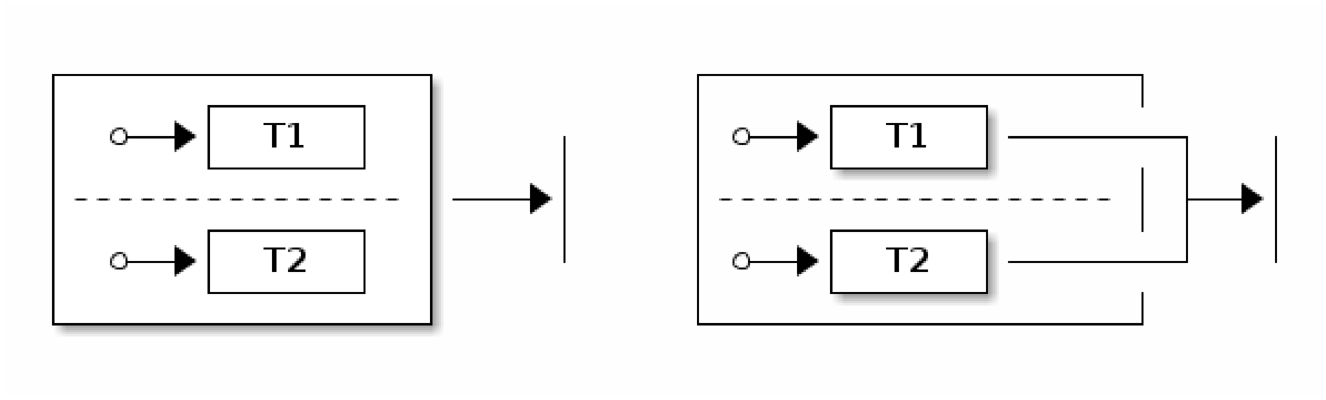
Fork pseudostate can be used to do an explicit entry into one or more regions.



Target state can be a parent state hosting regions, which simply means that regions are activated by entering its initial states. It’s also possible to add targets directly to any state in a region which allows more controlled entry into a state.

Join

Join pseudostate is used to merge several transitions together originating from different regions. It is generally used to wait and block for participating regions to get into its join target states.



Source state can be a parent state hosting regions, which means that join states will be a terminate states of a participating regions. It’s also possible to define source states to be any state in a regions which allows controlled exit from a regions.

Guard Conditions

Guard conditions are expressions which evaluates either to **TRUE** or **FALSE** based on extended state variables and event parameters. Guards are used with actions and transitions to dynamically choose if particular action or transition should be executed. Aspects of guards, event parameters and extended state variables are simply to make state machine design much more simple.

Events

Event is the most used trigger behaviour to drive a state machine. There are other ways to trigger behaviour to happen in state machine like a timer but events are the ones which really allows user to interact with a state machine. Events are also called as signals to possibly alter a state machine state.

Transitions

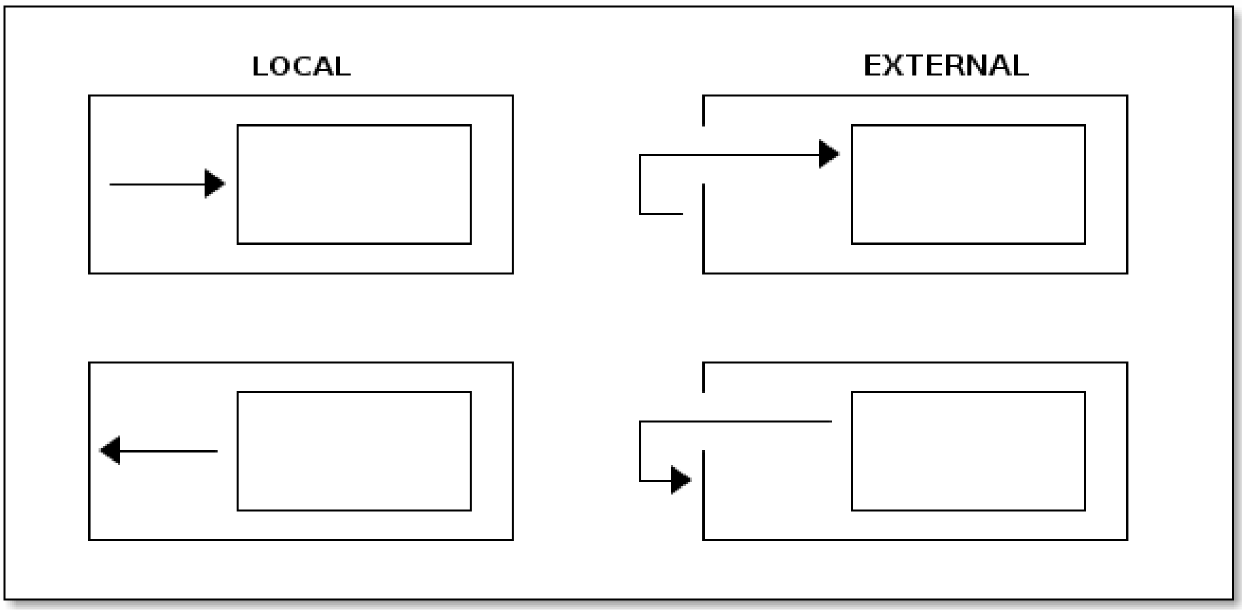
A transition is a relationship between a source state and a target state. A switch from a state to another is a *state transition* caused by a *trigger*.

Internal Transition

Internal transition is used when action needs to be executed without causing a state transition. With internal transition source and target state is always a same and it is identical with self-transition in the absence of state entry and exit actions.

External vs. Local Transition

Most of the cases external and local transition are functionally equivalent except in cases where transition is happening between super and sub states. Local transition doesn't cause exit and entry to source state if target state is a substate of a source state. Other way around, local transition doesn't cause exit and entry to target state if target is a superstate of a source state.



Above image shows a different between local and external transitions with a very simplistic super and sub states.

Actions

Actions are the ones which really glues state machine state changes with a user's own code. State machine can execute action on various changes and steps in a state machine like entering or exiting a state, or doing a state transition.

Actions usually have access to a state context which gives running code a choice to interact with a state machine in a various ways. State context i.e. is exposing a whole state machine so user can access extended state variables, event headers if transition is based on an event, or actual transition where it is possible to see more detailed where this state change is coming from and where it is going.

Hierarchical State Machines

Concept of a hierarchical state machine is used to simplify state design when particular states can only exist together.

Hierarchical states are really an innovation in UML state machine over a traditional state machines like Mealy or Moore machines. Hierarchical states allows to define some level of abstraction is a sense how java developer would define a class structure with abstract classes. For example having a nested state machine user is able to define transition on a multiple level of states possibly with a different conditions. State machine will always try to see if current state is able to handle an event together with a transition guard conditions. If these conditions are not evaluated to true, state machine will simply see what a super state can handle.

Regions

Regions which are also called as orthogonal regions are usually viewed as exclusive OR operation applied to a states. Concept of a region in terms of a state machine is usually a little difficult to understand but things gets a little simpler with a simple example.

Some of us have a full size keyboard with main keys on a left side and numeric keys on a right side. You’ve probably noticed that both sides really have their own state which you see if you press a numlock key which only alters behaviour of numbad itself. If you don’t have a full size keyboard you can buy a simple external usb numbad having only numbad part of a keys. If left and right side can freely exist without the other they must have a totally different states which means they are operating on different state machines.

It would be a little inconvenient to handle two different state machines as totally separate entities because in a sense they are still working together in a sense. This is why orthogonal regions can combine together a multiple simultaneous states within a single state in a state machine.

Appendix C. Distributed State Machine Technical Paper

This appendix provides more detailed technical documentation about using a Zookeeper with a Spring State Machine.

C.1 Abstract

Introducing a `distributed state` on top of a single state machine instance running on a single `jvm` is a difficult and a complex topic. `Distributed State Machine` is introducing a few relatively complex problems on top of a simple state machine due to its run-to-completion model and generally because of its single thread execution model, though orthogonal regions can be executed parallel. One other natural problem is that a state machine transition execution is driven by triggers which are either `event` or `timer` based.

Distributed `Spring State Machine` is trying to solve problem of spanning a generic `State Machine` through a jvm boundary. Here we show that a generic `State Machine` concepts can be used in multiple `jvm’s` and `Spring Application Contexts`.

We found that if `Distributed State Machine` abstraction is carefully chosen and backing distributed state repository guarantees `CP` readiness, it is possible to create a consistent state machine which is able to share distributed state among other state machines in an ensemble.

Our results demonstrate that distributed state changes are consistent if backing repository is `CP`. We anticipate our distributed state machine to provide a foundation to applications which need to work with a shared distributed states. This model aims to provide a good methods for cloud applications to have much easier ways to communicate with each others without having a need to explicitly build these distributed state concepts.

C.2 Intro

Spring State Machine is not forced to use a single threaded execution model because once multiple regions are uses, regions can be executed parallel if necessary configuration is applied. This is an important topic because once user wants to have a parallel state machine execution it will make state changes faster for independent regions.

When state changes are no longer driven by a trigger in a local jvm or local state machine instance, transition logic needs to be controlled externally in an arbitrary persistent storage. This storage needs to have a ways to notify participating state machines when distributed state is changed.

CAP Theorem states that "it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees, `consistency`, `availability` and `partition tolerance` ". What this means is that whatever is chosen for a backing persistence storage is it advisable to be `CP`. In this context `CP` means `consistency` and `partition tolerance`. Naturally `Distributed Spring Statemachine` doesn’t care about what is its `CAP` level but in reality `consistency` and `partition tolerance` are more important than `availability`. This is an exact reason why i.e. `Zookeeper` is a `CP` storage.

All tests presented in this article are accomplished by running custom `jepsen` tests in a following environment:

- Cluster having nodes n1, n2, n3, n4 and n5.
- Each node have a `Zookeeper` instance constructing an ensemble with all other nodes.
- Each node have a Chapter 34, *Web* sample installed which will connect to a local `Zookeeper` node.
- Every state machine instance will only communicate with a local `Zookeeper` instance. While connecting machine to multiple instances is possible, it is not used here.
- All state machine instances when started will create a `StateMachineEnsemble` using `Zookeeper` ensemble.
- Sample contains a custom rest api's which `jepsen` will use to send events and check particular state machine statuses.

All jepsen tests for `Spring Distributed Statemachine` are available from Jepsen Tests.

C.3 Generic Concepts

One design decision of a `Distributed State Machine` was not to make individual `State Machine` instance aware of that it is part of a `distributed ensemble`. Because main functions and features of a `StateMachine` can be accessed via its interface, it makes sense to wrap this instance using a `DistributedStateMachine`, which simply intercepts all state machine communication and collaborates with an ensemble to orchestrate distributed state changes.

One other important concept is to be able to persist enough information from a state machine order to reset a state machine state from arbitrary state into a new deserialized state. This is naturally needed when a new state machine instance is joining with an ensemble and it needs to synchronize its own internal state with a distributed state. Together with using concepts of distributed states and state persisting it is possible to create a distributed state machine. Currently only backing repository of a `Distributed State Machine` is implemented using a `Zookeeper`.

As mentioned in Chapter 23, *Using Distributed States* distibuted states are enabled by wrapping an instance of a `StateMachine` within a `DistributedStateMachine`. Specific `StateMachineEnsemble` implementation is `ZookeeperStateMachineEnsemble` providing integration with a `Zookeeper`.

C.4 ZookeeperStateMachinePersist

We wanted to have a generic interface `StateMachinePersist` which is able to persist `StateMachineContext` into an arbitrary storage and `ZookeeperStateMachinePersist` is implementing this interface for a `Zookeeper`.

C.5 ZookeeperStateMachineEnsemble

While distributed state machine is using one set of serialized contexts to update its own state, with zookeeper we're having a conceptual problem how these context changes can be listened. We're able to serialize context into a zookeeper `znode` and eventually listen when `znode` data is modified. However `Zookeeper` doesn't guarantee that you will get notification for every data change because registered `watcher` for a `znode` is disabled once it fires and user need to re-register that `watcher`. During this short time a `znode` data can be changed thus resulting missing events. It is actually very easy to miss these events by just changing data from a multiple threads in a concurrent manner.

Order to overcome this issue we're keeping individual context changes in a multiple `znodes` and we just use a simple integer counter to mark which `znode` is a current active one. This allows us to replay missed events. We don't want to create more and more znodes and then later delete old ones, instead we're using a simple concept of a circular set of znodes. This allows to use predefined set of znodes where a current can be determided with a simple integer counter. We already have this counter by tracking main `znode` data version which in `Zookeeper` is an integer.

Size of a circular buffer is mandated to be a power of two not to get trouble when interger is going to overflow thus we don't need to handle any specific cases.

C.6 Distributed Tolerance

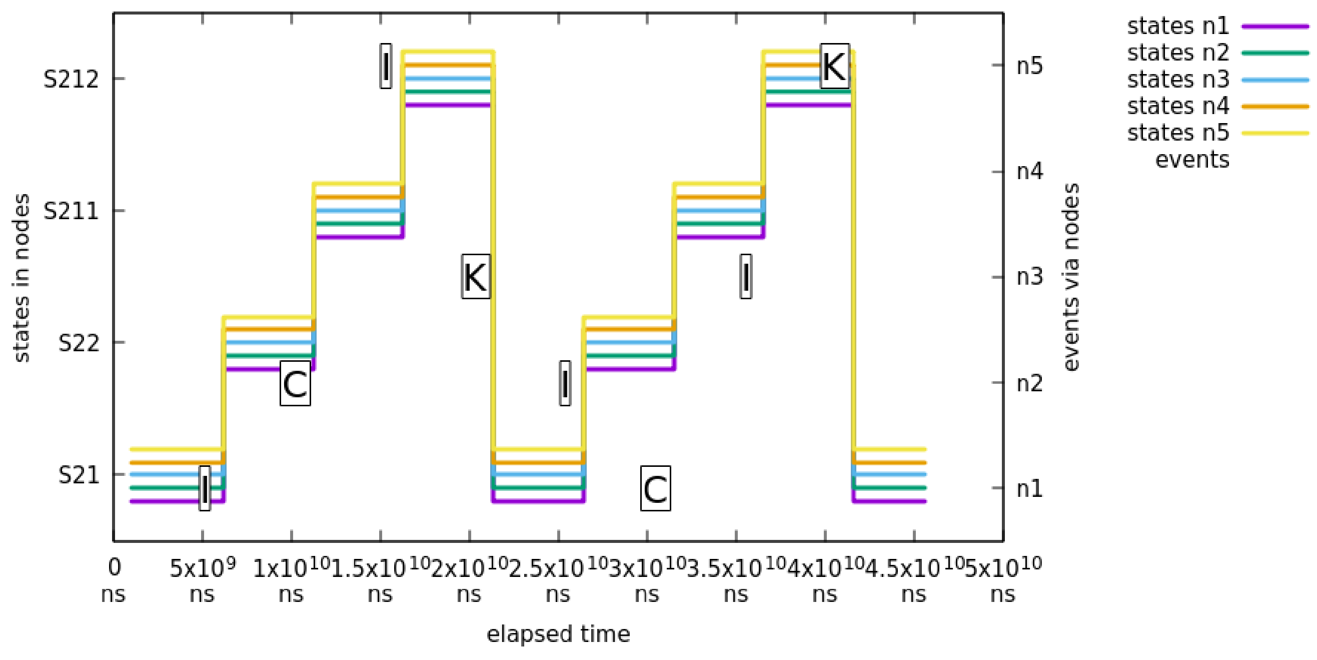
Order to show how a various distributed actions against a state machine work in a real life, we're using a set of `jepsen` tests to simulate various conditions which may happen in a real distributed cluster. These include a `brain split` on a network level, parallel events with a multiple `distributed state machines` and changes in an `extended state variables`. Jepsen tests are based on a sample Chapter 34, *Web* where this sample instance is run on multiple hosts together with a `Zookeeper` instance on every node where state machine is run. Essentially every state machine sample will connect to local `Zookeeper` instance which allows use, via `jepsen` to simulate network conditions.

Plotted graphs below in this chapter contain states and events which directly maps to a state chart which can be found from Chapter 34, *Web*.

Isolated Events

Sending an isolated single event into exactly one state machine in an ensemble is the most simplest testing scenario and demonstrates that a state change in one state machine is properly propagated into other state machines in an ensemble.

In this test we will demonstrate that a state change in one machine will eventually cause a consistent state change in other machines.



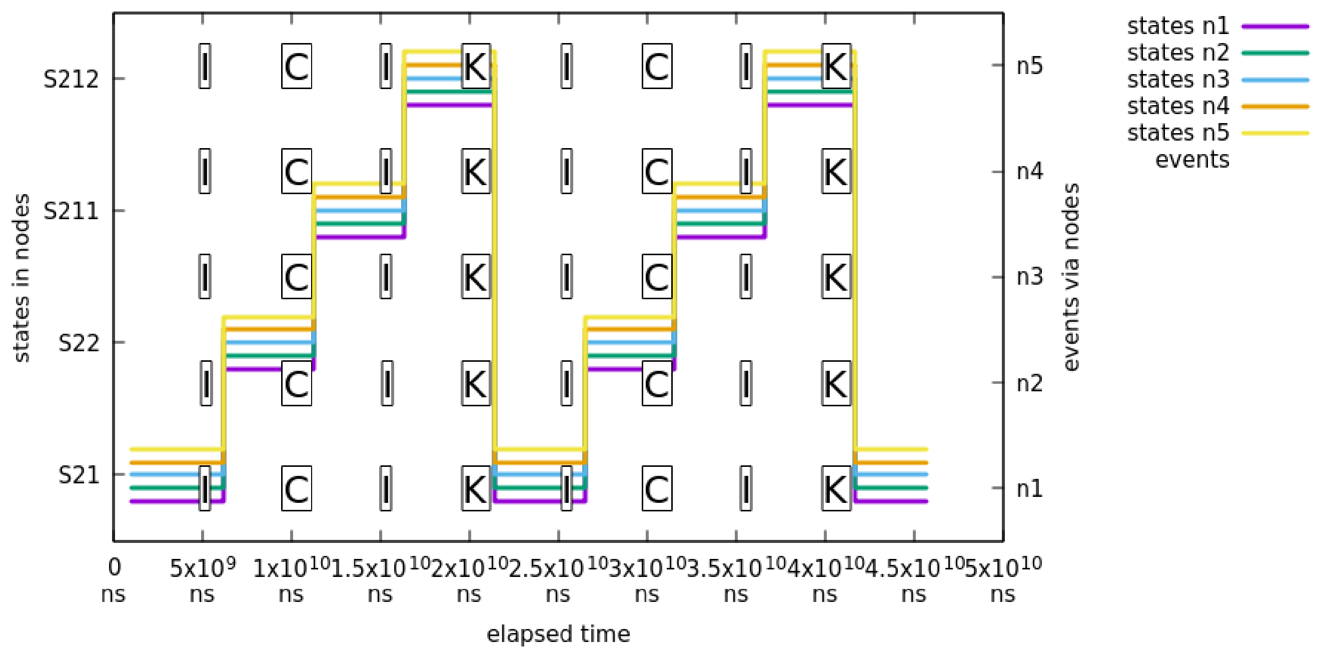
What's happening in above chart:

- All machines report state `S21`.
- Event `I` is sent to node `n1` and all nodes report state change from `S21` to `S22`.
- Event `C` is sent to node `n2` and all nodes report state change from `S22` to `S211`.
- Event `I` is sent to node `n5` and all nodes report state change from `S211` to `S212`.
- Event `K` is sent to node `n3` and all nodes report state change from `S212` to `S21`.
- We cycle events `I`, `C`, `I` and `K` one more time via random nodes.

Parallel Events

Logical problem with multiple distributed state machines is that if a same event is sent into a multiple state machine exactly at a same time, only one of those events will cause a distributed state transitions. This is somewhat expected scenario because a first state machine, for this event, which is able to change a distributed state will control the distributed transition logic. Effectively all other machines receiving this same event will silently discard the event because distributed state is no longer in a state where particular event can be processed.

In this test we will demonstrate that a state change caused by a parallel events throughout an ensemble will eventually cause a consistent state change in all machines.



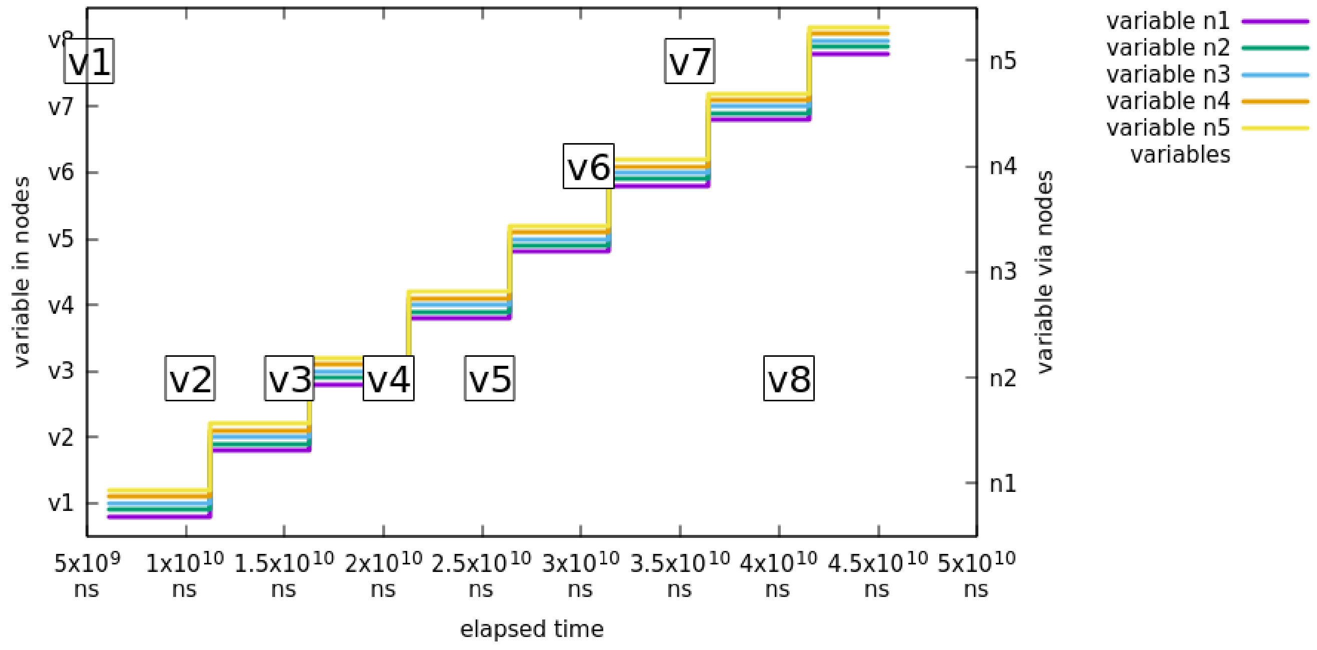
What's happening in above chart:

- We use exactly same event flow than in previous sample the section called "Isolated Events" with a difference that events are always sent to all nodes.

Concurrent Extended State Variable Changes

Extended state machine variables are not guaranteed to be atomic at any given time but after a distributed state change, all state machines in an ensemble should have a synchronized extended state.

In this test we will demonstrate that a change in extended state variables in one distributed state machine will eventually be consistent in all distributed state machines.



What’s happening in above chart:

- Event `J` is send to node `n5` with event variable `testVariable` having value `v1`. All nodes are then reporting having varible `testVariable` as value `v1`.
- Event `J` is repeated from variable `v2` to `v8` doing same checks.

Partition Tolerance

We need to always assume that sooner or later things in a cluster will go bad whether it is just a crash of a `Zookeeper` instance, a state machine or a network problem like a `brain split`. Brain split is a situation where existing cluster members are isolated so that only part of a hosts are able to see each others. Usual scenario is that a brain split will create a minority and majority partitions of an ensemble where hosts in a minority cannot participate in an ensemble anymore until network status has been healed.

In below tests we will demonstrate that various types of brain-split’s in an ensemble will eventually cause fully synchronized state of all distributed state machines.

There are two scenarios having a one straight brain split in a network where where `Zookeeper` and `Statemachine` instances are split in half, assuming each `Statemachine` will connect into a local `Zookeeper` instance:

- If current zookeeper leader is kept in a majority, all clients connected into majority will keep functioning properly.
- If current zookeeper leader is left in minority, all clients will disconnect from it and will try to connect back till previous minority members has successfully joined back to existing majority ensemble.

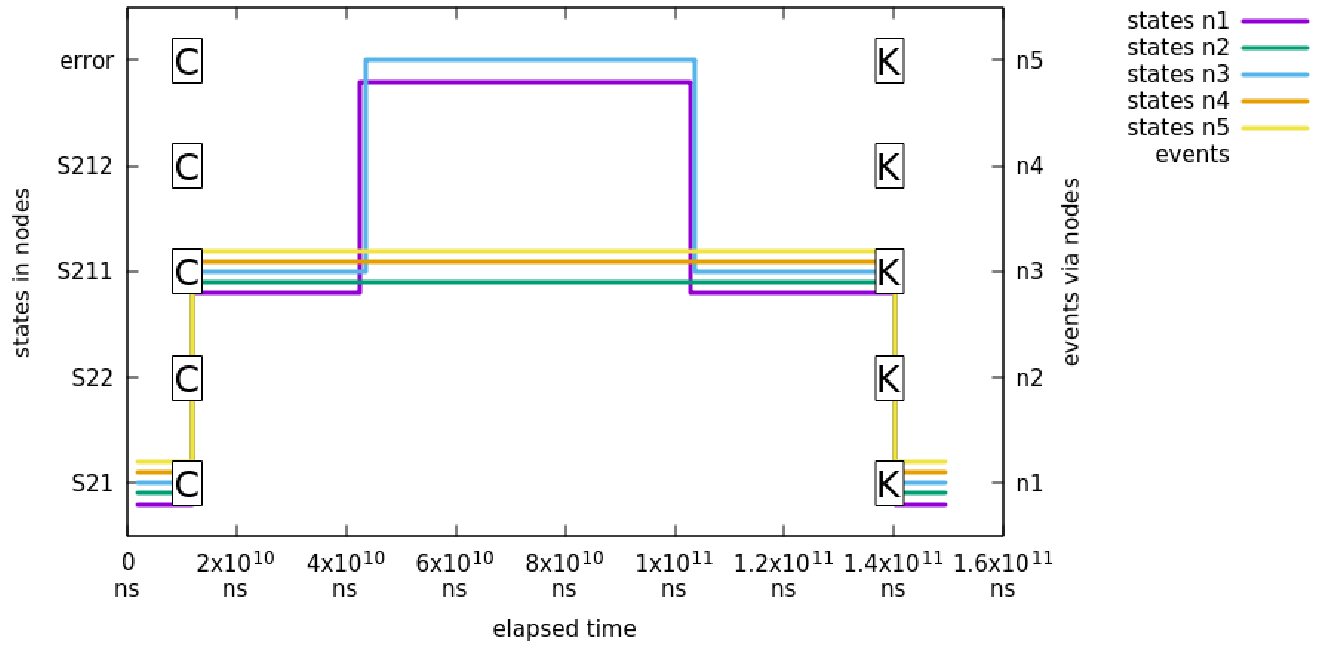
Note

In our current `jepsen` tests we can’t separate zookeeper split brains scenarios between leader left in majority or minority so we need to run tests multiple time to accomplish this situation.

Note

In below plots we have mapped a state machine error state into an `error` to indicate that `state machine` is in error state instead or a normal state. Please indicate this when interpreting chart states.

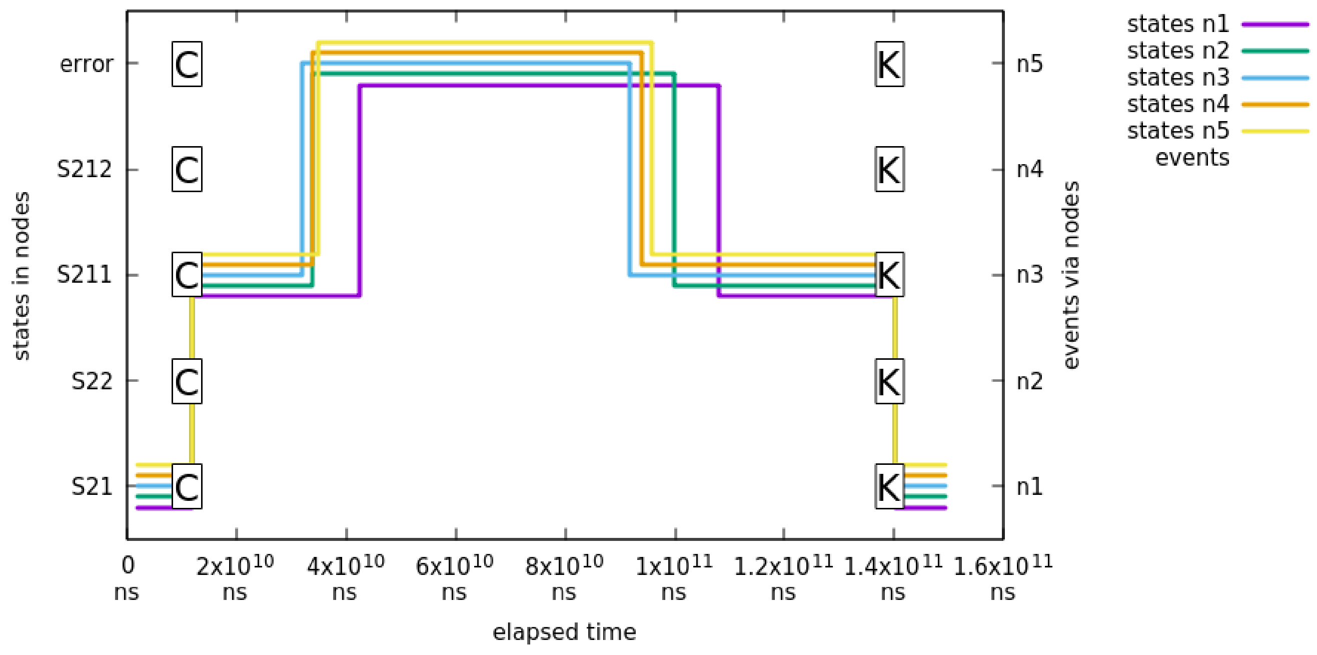
In this first test we show that when existing zookeeper leader was kept in majority, 3 out of 5 machines will continue as is.



What’s happening in above chart:

- First event `C` is sent to all machine leading a state change to `S211`.
- Jepsen nemesis will cause a brain-split which is causing partitions of `n1/n2/n5` and `n3/n4`. Nodes `n3/n4` are left in minority and nodes `n1/n2/n5` construct a new healthy majority. Nodes in majority will keep function without problems but nodes in minority will get into error state.
- Jepsen will heal network and after some time nodes `n3/n4` will join back into ensemble and synchronize its distributed status.
- Lastly event `K1` is sent to all state machines to ensure that ensemble is working properly. This state change will lead back to state `S21`.

In this second test we show that when existing zookeeper leader was kept in minority, all machines will error out:

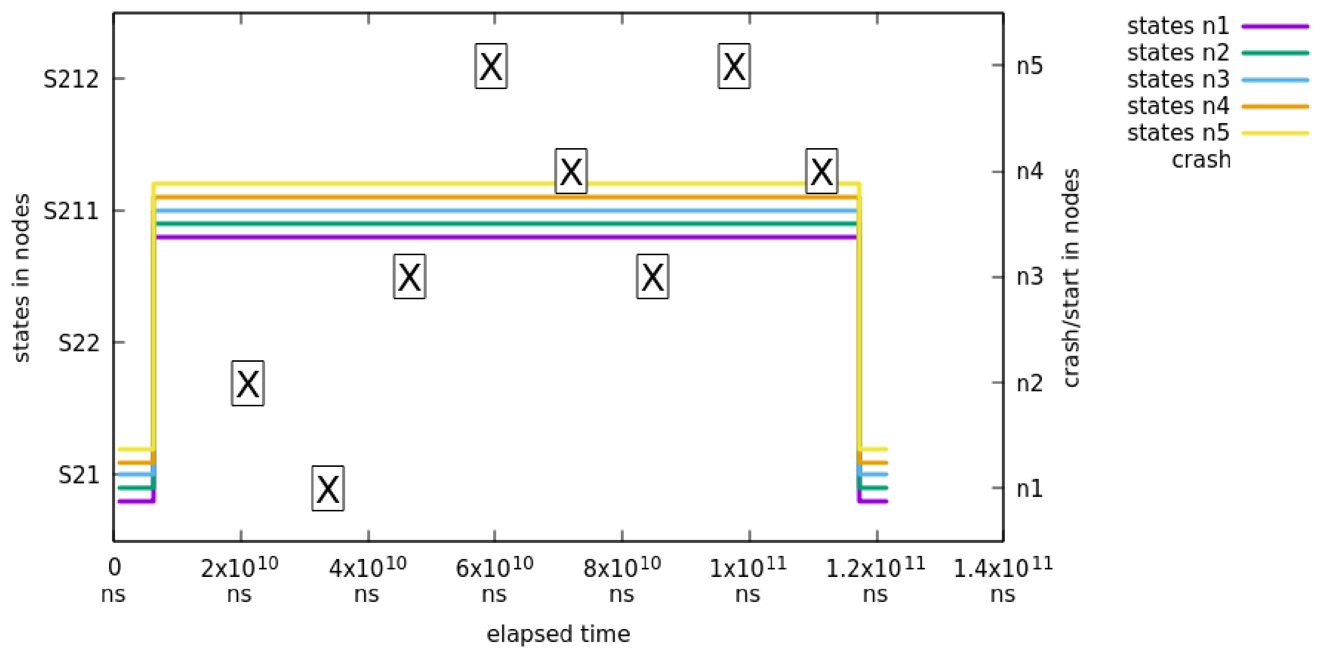


What’s happening in above chart:

- First event `C` is sent to all machine leading a state change to `S211`.
- Jepsen nemesis will cause a brain-split which is causing partitions so that existing `Zookeeper` leader is kept in minority and all instances are disconnected from ensemble.
- Jepsen will heal network and after some time all nodes will join back into ensemble and synchronize its distributed status.
- Lastly event `K1` is sent to all state machines to ensure that ensemble is working properly. This state change will lead back to state `S21`.

Crash and Join Tolerance

In this test we will demonstrate that killing existing state machine and then joining new instance back into an ensemble will keep the distributed state healthy and newly joined state machines will synchronize their states properly.



Note

In this test, states are not checked between first X and last X, thus graph will will show flat line in between. States are checked exactly where state change happens between S21 and S211.

What's happening in above chart:

- All state machines are transitioned from initial state S21 into S211 so that we can test proper state synchronize during join.
- X is marking when a specific node has been crashed and started.
- At a same time we request states from all machines and plot it.
- Finally we do a simple transition back to S21 from S211 to make sure that all state machines are still functioning properly.