

# **Flood - User Guide**

Kajetan Rzepecki

September 24, 2013

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Use cases . . . . .	3
1.2	Supported Protocols . . . . .	3
1.3	Dependencies . . . . .	3
<b>2</b>	<b>Inner workings</b>	<b>4</b>
2.1	Simulated Users . . . . .	4
2.2	User sessions . . . . .	4
2.2.1	Session selection . . . . .	4
2.2.2	Session inheritance . . . . .	4
2.2.3	Actions & Event handlers . . . . .	4
2.2.4	Timers & Counters . . . . .	4
2.3	Flood phases . . . . .	4
<b>3</b>	<b>Flood scenarios</b>	<b>5</b>
3.1	Scenario file . . . . .	5
3.2	Server setup . . . . .	5
3.3	Phases setup . . . . .	5
3.4	User session setup . . . . .	7
3.5	User actions . . . . .	8
3.6	Metadata . . . . .	12
3.7	Example scenarios . . . . .	12
<b>4</b>	<b>Flood results</b>	<b>13</b>
4.1	Results format . . . . .	13
4.2	Goal schemas . . . . .	13
4.3	Continuous Integration integration . . . . .	13
4.4	Example results . . . . .	13

# 1 Introduction

**Flood** is a load simulator useful for automatic **Comet/PUSH application** stress-testing. It is **asynchronous, event based** and enables you to create JSON encoded **test scenarios of arbitrary complexity** involving **tens of thousands of simulated users**, no Erlang required!

## 1.1 Use cases

Some of the most common use cases that **Flood** might be helpful in testing are:

- **Massive, real-time, on-line chats,**
- **Publisher-Subscriber channels,**
- **Instant messaging.**

However, Flood is general enough to test *any* event-based Comet application that uses the supported protocols.

## 1.2 Supported Protocols

**Flood** currently supports the **Socket.IO** protocol over **WebSocket** and **XHR-polling** transports with emphasis on Socket.IO event based communication. Flood also has *some* capabilities of using **raw HTTP** requests.

## 1.3 Dependencies

Flood uses several awesome libraries that are listed below. Since Flood is currently in development, *no particular stable versions are required* and by default the newest available versions are pulled in.

- `ibrowse` - an HTTP client, found [here](#).
- `lager` - a logging framework, found [here](#).
- `folsom` - a metrics system, found [here](#).
- `jsonx` - a fast JSON parser, found [here](#).
- `jesse` - a JSON Schema validator, found [here](#).
- `websocket_client` - a WebSocket client, found [here](#).

## 2 Inner workings

This section describes what happens behind the scenes in **Flood** and how it reflects its usage.

### 2.1 Simulated Users

- FSMs
- State transitions

### 2.2 User sessions

#### 2.2.1 Session selection

- Roulette algorithm

#### 2.2.2 Session inheritance

- Single inheritance ordering.
- Multiple inheritance ordering.
- Why so OOP?

#### 2.2.3 Actions & Event handlers

- `onsocketio`
- `onevent`
- `ontimeout`

#### 2.2.4 Timers & Counters

- Starting/stopping/restarting timers
- Managing counters

### 2.3 Flood phases

- Phases purpose
- Phase goals

## 3 Flood scenarios

This section describes the Flood scenario files and gives some general guidelines for writing them. Example scenarios can be found [here](#).

### 3.1 Scenario file

**Flood** uses JSON to encode test scenarios, no Erlang is required. Each scenario resides in a separate file and optionally several goal files (described in detail later). The overall structure of a Flood scenario consists of three required sections:

```
{
  "server" : {
    // Server setup.
  },

  "phases" : {
    // Test phases & goals.

    "phase_I" : {
      ...
    },
    ...
  },

  "sessions" : {
    // User session descriptions.

    "session_A" : {
      ...
    },
    ...
  }
}
```

### 3.2 Server setup

The **server** section is rather straightforward; it is used to setup the server connection. It has to define several mandatory fields:

```
"server" : {
  "host" : "",           // The server host.
  "port" : 0,           // The server port.
  "endpoint" : "",      // Endpoint used to connect to.
  "metadata" : {}      // Server-wide metadata (optional).
}
```

Example server configuration that will cause Flood to connect to `http://localhost:80/socket.io/1/` and define some server-wide metadata (more on metadata can be found [here](#)):

```
"server" : {
  "host" : "localhost",
  "port" : 80,
  "endpoint" : "/socket.io/1/",
  "metadata" : {
    "foo" : "bar"
  }
}
```

### 3.3 Phases setup

The **phases** section may define several arbitrarily named Flood phases. The ordering does not matter, as each phase explicitly names its start time.

```

"phases" : {
  "A" : {
    // A's description.
  },

  "B" : {
    // B's description.
  },
  ...
}

```

Each phase description has to follow this format:

```

"phase_I" : {
  "users" : 0,           // Number of users spawned during this phase.
  "user_sessions" : [], // Sessions spawned users should follow.

  "start_time" : 0,      // Time (in milliseconds) at which to start this phase.
  "spawn_duration" : 0,  // Duration (in milliseconds) Flood should take to spawn the users.

  "goal" : {},           // Goal of this phase (optional).
  "test_interval" : 0,   // Interval (in milliseconds) of the goal checks (optional).
  "timeout" : 0,         // Timeout (in milliseconds) of this phase (optional).

  "metadata" : {}        // Phase-wide metadata (optional).
}

```

The meaning of each of the fields is as follows:

- **users** - an integer number of users spawned during this phase. It is **mandatory**.
- **user\_sessions** - a array of Flood user session names; the concrete user session will be selected at **random according to a sessions weight** (more about this can be found here). It is **mandatory**.
- **start\_time** - an integer value that names a point in time (**in milliseconds**), relative to the start of the Flood, at which a phase should be started. It is **mandatory**.
- **spawn\_duration** - an integer value that tells Flood how much time (**in milliseconds**) it should take to spawn **users** number of users. Users are spawned uniformly throughout this duration. Keep in mind that for various performance related reasons Flood **may actually take longer** to spawn the users, however it will never take less time to do so. This field is **mandatory**.
- **goal** - either an arbitrary JSON term that is a description of the goal of this phase (more on goals can be found here) or a string containing a path to the file containing the goal description relative to scenario file. This field is **optional**; not defining it will result in no goal checking whatsoever.
- **test\_interval** - an integer value that tells Flood at what intervals (**in milliseconds**) in should check whether the goal has been reached. It is **optional**; not defining it will result in a single check at the phase timeout.
- **timeout** - an integer value that names a point in time (**in milliseconds**), relative to the start of the Flood, at which a phase should be terminated if it is still running. It is **optional**.
- **metadata** - a JSON object defining some phase-wide metadata (more on metadata later). It is **optional**.

Example phases setup:

```

"phases" : {
  "phase_I" : {
    "metadata" : { },

    "users" : 1000,
    "user_sessions" : ["session_A", "session_B"],

    "start_time" : 1000,

```

```

    "spawn_duration" : 1000
  },

  "phase_II" : {
    "metadata" : { },

    "users" : 1000,
    "user_sessions" : ["session_C"],

    "start_time" : 2000,
    "spawn_duration" : 5000

    "goal" : "./goal.jsonschema",
    "test_interval" : 100,
    "timeout" : 10000
  }
}

```

This setup will schedule two Flood phases. The first phase, **phase\_I**, will start at 1000 ms and spawn 1000 users following either **session\_A** or **session\_B** over 1000 ms duration. The second phase, **phase\_II**, will start at 2000 ms and spawn 1000 users following **session\_C** over 5000 ms duration. Additionally, a **phase\_II** goal check will be scheduled every 100 ms starting at 2000 ms and running until the goal provided in “./goal.jsonschema” file is met or until the phase timeout, set at 10000 ms, is reached.

### 3.4 User session setup

The **sessions** section may define several arbitrarily named Flood user sessions. The ordering does not matter, as each session explicitly names its relations to other sessions.

```

"sessions" : {
  "session_A" : {
    // A's description.
  },

  "session_B" : {
    // B's description.
  },
  ...
}

```

Each session description has to follow this format:

```

"session_A" : {
  "extends" : [], // Array of sessions extended by this session (optional).

  "weight" : 0.0, // Weight of this session (optional).
  "transport" : "", // Socket.IO transport used by this session (optional).

  "metadata" : {}, // Session-wide metadata (optional).
  "do" : [] // Array of actions to be performed by the user (optional).
}

```

The meaning of each of the fields is as follows:

- **extends** - an array of session names that this session extends (more about session inheritance can be found here). It is **optional** and omitting it means that this session does not extend any other sessions.
- **weight** - a real number determining how often simulated users will choose this session over other sessions (more on session selection can be found here); it is completely relative and depends on the total weight of a subset of sessions considered at one point (for examples at a certain Flood phase’s startup). It is **optional** and defaults to **0.0**.
- **transport** - a string naming a Socket.IO compatible transport protocol. It should be either of **websocket** or **xhr-polling**, but in general it is **optional** and defaults to the empty string.

- **metadata** - a JSON object defining some session-wide metadata (more on metadata later). It is **optional**.
- **do** - an array of actions to be performed by the users following this session (more on actions & event handlers can be found here; a list of all available actions can be found in the next section). It is **optional** and defaults to the empty array.

### 3.5 User actions

Actions are performed by the simulated users after their initialization and whenever an event triggers an event handler (for example, a Socket.IO message is received or a timer is due). Actions **ordering does matter** as some actions change the state of the simulated users.

Actions are represented as short JSON arrays consisting of an **action\_ID** and a JSON object listing actions arguments:

```
["action_ID", {
  "argument_1" : "value_1", // Argument ordering does not matter.
  "argument_2" : "value_2",
  ...
}]
```

For convenience, some actions define a shorter forms that mean exactly the same, for example:

```
["action_ID", "value_1", "value_2"] // Mind the arguments ordering.
```

The following list lists available actions, describes their effects and arguments, and gives an example invocation in both full and short forms:

- **inc** - increments a named counter either by 1 or by **Value**. Example usage:

```
["inc", "counter_name"]
["inc", "counter_name", Value]
["inc", {
  "name" : "counter_name",
  "value" : Value
}]
```

- **dec** - decrements a named counter either by 1 or by **Value**. Example usage:

```
["dec", "counter_name"]
["dec", "counter_name", Value]
["dec", {
  "name" : "counter_name",
  "value" : Value
}]
```

- **set** - sets a named counter to a given **Value**. Example usage:

```
["set", "counter_name", Value]
["set", {
  "name" : "counter_name",
  "value" : Value
}]
```

- **start\_timer** - starts a named timer timeouting in **Timeout** milliseconds. Example usage:

```
["start_timer", "timer_name", Timeout]
["start_timer", {
  "name" : "timer_name",
  "time" : Timeout
}]
```

- **stop\_timer** - stops a named timer preventing it from timing out and triggering an event dispatch. Example usage:



```
["stop_timer", "timer_name"]
["stop_timer", {
  "name" : "timer_name"
}]
```

- `restart_timer` - restarts a named timer. Essentially, performs `stop_timer` and `start_timer` is quick succession. Example usage:

```
["restart_timer", "timer_name", Timeout]
["restart_timer", {
  "name" : "timer_name",
  "time" : Timeout
}]
```

- `timed` - executes a set of actions while timing their execution time which it then stores in a named counter. Results in whatever the actions result in. Example usage:

```
["timed", {
  "name" : "counter_name",
  "do" : [
    Action,
    ...
  ]
}]
```

- `on_timeout` - adds several timeout handlers to the simulated users state. If a given timeout handler already exists, new actions are appended **after** the existing ones, meaning they will be executed after the existing actions. Example usage:

```
["on_timeout", {
  "timer_name_1" : [
    Action,
    ...
  ],
  ...
}]
```

- `on_event` - adds several event handlers to the simulated users state. If a given event handler already exists, new actions are appended **after** the existing ones, meaning they will be executed after the existing actions. Example usage:

```
["on_event", {
  "event_1" : [
    Action,
    ...
  ],
  ...
}]
```

- `on_socketio` - adds several messages handlers to the simulated users state. If a given message handler already exists, new actions are appended **after** the existing ones, meaning they will be executed after the existing actions. Example usage:

```
["on_socketio", {
  "opcode_1" : [
    Action,
    ...
  ],
  ...
}]
```

- `emit_event` - emits Event with Args as a Socket.IO message with the event opcode. Example usage:

```
["emit_event", {
  "name" : Event,
  "args" : Args
}]
```

- **emit\_socketio** - emits a Socket.IO message to the given **Endpoint** with the given **Opcode** and **Payload**. Example usage:

```
["emit_socketio", {
  "opcode" : Opcode,
  "endpoint" : Endpoint,
  "data" : Payload
}]
```

- **emit\_http** - emits a synchronous HTTP request with a given **Method**, **Body**, **Headers** and **Timeout** to a given **Url**. Afterwards, executes actions defined in **on\_reply** or **on\_error** when the requests succeeded or failed respectively. Example usage:

```
["emit_http", {
  "url" : Url,
  "method" : Method,
  "body" : Body,
  "headers" : Headers,
  "timeout" : Timeout,

  "on_reply" : [
    Action,
    ...
  ],

  "on_error" : [
    Action,
    ...
  ]
}]
```

- **match** - performs either a JSON-based or RegExp-based pattern-matching operation on **Subject**. RegExp-based matching takes precedence over JSON-based matching. The results are stored in the simulated users metadata under **Name\_#** (where **#** is the index of the match) for RegExp-based matching or under respective **\$names** for JSON-based matching. Afterwards, executes actions defined in either **on\_match** or **on\_nomatch** when the matching succeeds or fails respectively. Example usage:

```
["match", {
  "name" : Name,
  "subject" : Subject,
  "re" : "regexp",

  "on_match" : [
    Action,
    ...
  ],

  "on_nomatch" : [
    Action,
    ...
  ]
}]
```

```
["match", {
  "subject" : Subject,
  "json" : {
    "field_1" : "$value_1",
```

```

        "field_2" : "$value_2",
        ...
    },

    "on_match" : [
        Action,
        ...
    ],

    "on_nomatch" : [
        Action,
        ...
    ]
}
}]

```

- **case** - performs a value case dispatch on a given **Value** selecting a matching **Branch** and executing its respective actions. Example usage:

```

["case", Value, {
    Branch : [
        Action,
        ...
    ],
    ...
}]
["case", {
    "condition" : Value,
    "branches" : {
        Branch : [
            Action,
            ...
        ],
        ...
    }
}]

```

- **def** - adds new metadata to the simulated users state. Example usage:

```

["def", {
    "key_1" : "value_1",
    "key_2" : "value_2",
    ...
}]

```

- **terminate** - immediately stops actions execution and terminates the simulated user with termination reason set to **Reason**. Disconnects him from the server and terminates his process. Example usage:

```

["terminate", Reason]
["terminate", {
    "reason" : Reason
}]

```

- **log** - prints a log line to the console formatting it with the **Format** and **Values**. The **Format** format is the same as Erlangs `io:format/2` (why yes, I did lie about the “no Erlang required” thing, deal with it). Example usage:

```

["log", Format, Values]
["log", {
    "format" : Format,
    "values" : Values
}]

```

- **!log** - a convenience action that allows easy **log** toggling; does nothing. Example usage:

```
[!log", Format, Values]
[!log", {
    "format" : Format,
    "values" : Values
}]
```

### 3.6 Metadata

- Metadata ordering
- Introducing new metadata
- JSON Substitutions

### 3.7 Example scenarios

- Sessions
- Single ping
- Continuous ping
- “3rd party” requests

## **4 Flood results**

### **4.1 Results format**

- JSON structure
- Counters
- Timers
- Available statistics

### **4.2 Goal schemas**

- JSON Schema structure
- Testing intervals
- Reaching goals
- Goal timeouts

### **4.3 Continuous Integration integration**

- Running Flood automagically

### **4.4 Example results**

- Sessions
- Single ping
- Continuous ping
- “3rd party” requests