AsyncMachine
The Definitive Guide

...WORK IN PROGRESS...

Introduction

AsyncMachine is a modern state manager with intelligence built it.

It's a loose combination of several different concepts, while still being a pretty minimalistic library. It was born slowly over the years of writing complex JavaScript apps and other asynchronous systems. Most of all, it's a state machine, thus the label *Hybrid State Machine*.

The project borrows from the following concepts (warning - buzzwords ahead):

- Actor Model
- Declarative Dependency Graph
- Non-Deterministic State Machine
- Async Control Flow
- Declarative Scheduler
- State Manager
- Event Stream Processor
- Aspect Oriented Programming

Basic topics

AsyncMachine is best to be described by simply listing its components and features. Most of them will probably sound familiar.

Components:

- states
- transitions
- relations
- clocks
- pipes
- queues

Features:

- synchronous mutations
- negotiation
- cancellation
- exception handling

This guide includes many examples to make understanding of those concepts as easy as possible. They contain visual representation where applicable and each of them can be edited and inspected live on Stackblitz. Every method is linked to the API docs and method signatures are provided using TypeScript definitions. Almost all of the examples makes use of the machine factory, which imports have been skipped for readability purposes. Assume that every example starts with:

```
import { machine } from "asyncmachine";
```

It's also required to have the npm module installed:

```
npm install asyncmachine
```

States

Everything is a state

The basic idea of AsyncMachine is very simple - **everything is a state**. **State** represents certain abstraction in time. Classic example of a problem which can be easily solved using states is an elevator - it can be on each of the floors, it may be going down or up, it may have pushed buttons and it may also be stopped. Some of those states can happen simultaneously, others are mutually exclusive.

State definition

State definition is it's name and a JSON-like structure describing it's properties and relations (more about those later). Here's TypeScript interface:

```
interface IState {
  // relations
  add?: string[];
  drop?: string[];
  require?: string[];
  after?: string[];

  // properties
  auto?: boolean;
  multi?: boolean;
}
```

What to consider a "state"

State represents a model of your program's input, output and side effects. The fact that certain states are active or not is based on the processed data and external APIs. In case of the elevator example mentioned above, you probably don't want to make every passenger a state described by it's name, although it could be useful to have states like `Empty`, `Boarded` and `Overfilled`.

In short - **not all of you app's data should be considered "states"**, as those represent a higher-level view on what's happening in your program.

Asynchronous state

Considering that everything is a state, representing asynchrony can be achieved by creating two separate states - **in progress** and **done**. For example, when downloading a file, it would be `Downloading` and `Downloaded`. Having every meaningful action and event encapsulated as a state allows us to precisely react on input events. In case of the previous example, we could've had another states called `ButtonPressed`, which triggers the download, but makes a different decision in case the state `Downloading` is currently active. Same goes for `Downloaded` which can behave differently if there has been another `Downloading` state since the download process began.

Mutating the state

In AsyncMachine, **every state mutation is synchronous** and executed by a transition (but let's not worry about transitions just yet). This takes advantage of the event loop, which also is always synchronous during a single tick. Here are the most basic methods to alter the state of a machine:

Add

Add is the most popular method, as it preserves the currently active states.

```
class AM {
  add(states: string[] | string, ...params: any[]): boolean | null;
}
```

Example:

```
const example = machine(["A", "B", "C"]);
example.add("A");
example.is(); // -> ['A']
```

Drop

Drop de-activates only specified states.

```
class AM {
  drop(states: string[] | string, ...params: any[]): boolean | null;
}
```

Example:

```
const example = machine(["A", "B", "C"]);
example.add(["A", "B"]);
example.drop("A");
example.is(); // -> ['B']
```

Set

Set removes all currently active states and activates only the passed ones - could also be called `force`.

```
class AM {
  set(states: string[] | string, ...params: any[]): boolean | null;
}
```

Example:

```
const example = machine(["A", "B", "C"]);
example.add("A");
example.set("B");
example.is(); // -> ['B']
```

Additionally, **mutations aren't nested** - one can happen only after the first one finishes. The trick is a **tri-state logic** - after you try to mutate the state of a machine, you can get the following returns:

- `true` - successful
- `false` - rejected
- `null` - queued

You can check prior to mutating if the machine is busy executing another transition by using the `duringTransition()` method (more about that in the locks section).

Machine definition

The simplest way to define a state machine is to provide an object with state names assigned to a state definitions and use the `machine` factory, like so:

```
const state = {
  Wet: { require: ["Water"] },
  Dry: { drop: ["Wet"] },
  Water: { add: ["Wet"], drop: ["Dry"] }
};
const example = machine(state);
```

State names have predefined **naming convention** which is `CamelCase`. Thanks to that they are easily distinguishable from a handler's suffix, which follows the states name in `snake_case`. They also can't start with a number.

An important thing when defining a machine is to give it's an ID. This is optional as a random ID is always assigned, but helps reading the log and inspecting the state graph.

Example:

```
const example = machine(["A"]);
// the id() methos is chainable
example.id("example");
```

State clock

Every state has a clock, which increments every time state changes from being in-active to active. The purpose of the state clock is to distinguish different instances of the same state. Like in the example above about a button and a download process, the `Downloaded` state can check if the he has been originated by the current `Downloading` state. This prevents a common race condition in JavaScript called **double callback execution**.

The API for the clock is as trivial as `clock(state: string): number` . In most cases the calls to the clock are done by higher order functions, but being aware of the concept is crucial to understanding of how AsyncMachine works.

Checking the current state

Synchronous state checks are performed by the `is()` method. It serves both asserting a state and getting the current one. Consider the following example:

```
const example = machine(["A", "B", "C"]);
example.add("A");
example.is(); // -> ['A']
example.is("A"); // -> true
example.is("B"); // -> false
example.is("A", example.clock("A")); // -> true
```

First three calls are pretty straightforward, while the last one can be harder to understand. What it does is it asserts that the given state is currently in that specific tick (of it's clock). Usually you'd store the clock beforehand and check it later in the execution (eg in a callback).

Finite State Machine vs Multi State Machine

Pretty much everyone is more-or-less familiar with the idea of a Finite State Machine (FSM), either from college, books or the internet. It's one of the most basic and powerful tools in Computer Science. AsyncMachine isn't one of those. It can have many simultaneously active state and during a mutation triggers more than one transition (handler in this case). That makes it fall into the category of non-deterministic state machines and comparisons to FSM will not make it easier to understand the presented concepts.

Auto states

Auto states are one of the most powerful concepts of AsyncMachine. Those are simply the **states which after every mutation will try to activate themselves**, considering their dependencies are met. In a more technical description - after every transition, there can be another transition consisting of auto states only. If at least one of them gets accepted - the machine mutates again. Mentioned transition is prepended to the queue, that means it happens immediately after each mutation instead of being queued like manual mutations.

Multi states

Multi state describe a state which can be set many times. It always trigger "enter" and "state" transition handlers, plus the clock is always incremented. It's useful for describing many instances of the same event (eg a network input) without having to define more than one transition handler. Additionally, the incremented clock allows you to manually synchronize callback results.

Actions as states

A state usually describes something more "persistent" then an action, eg `Click` is an action while `Downloaded` is a real state. Those are of course very common and still can be represented in AsyncMachine. All that's needed is a self `drop` at the end of the final transition handler. Code example:

```
const example = machine(["Click"]);
example.Click_state = function() {
  this.drop("Click");
};
```

In this way the state of a machine stays integral and you can reference the action using state relations.

Transitions

A transition is a mutation of machine's active states to a different subset of possible states. Eg machine with states `['A', 'B', 'C']` can have state mutated from `['A']` to `['A', 'B']`. Each transition has several steps and (optionally) calls several handlers. Exception to this rule is the self transition handler and the multi states, which trigger a self transition handlers for a **mutation request to an already active state**.

Transition handlers

Each state can define five state handlers (`enter`, `exit`, `state`, `end` and `self`) and many transitional handlers (eg `A_B`) used when transitioning **to** or **from** other states.

List of handlers during a transition between `['A'] -> ['B']`, in the order of execution:

- `A_exit` - negotiation handler
- `A_B` - negotiation handler
- `A_any` - negotiation handler
- `any_B` - negotiation handler
- `B_enter` - negotiation handler
- `A_end` - final handler
- `B_state` - final handler

This list will be a little bit different for multi states and self transitions.

Self transition handler

Purpose of self transiton handlers is to be able to react to a mutation which has the same state **before and after the mutation**.

List of handlers during a transition between `['A'] -> ['A', 'B']`, in the order of execution:

- `any_B` - negotiation handler
- `B_enter` - negotiation handler
- `A_self` - final handler and a self handler
- `B_state` - final handler

Defining handlers

Transition handlers can be defined in two ways:

- As an object method - on the machine itself, or on the target object
- As an event listener

Example of both declaration types for an `A_enter` negotiation phase transition handler:

```
const example = machine(["A", "B"]);
// method handler
example.A_enter = function() {
  // handler's logic
};
// listener handler
example.on("A_enter", function() {
  // handler's logic
});
```

It's important to note, that when binding a listener using the event emitter API it can be **executed immediately** in case:

- `A_enter` and `A_state` when the state `A` is currently active
- `A_exit` and `A_end` when the state `A` is currently in-active

Example:

```
const example = machine(["A"]);
example.add(["A"]);
example.on("A_enter", function() {
  console.log("executed");
});
// will print 'executed'
```

Passing params

When you pass parameters to one of the mutation methods you can also provide additional parameters, besides the names of the requested states. Those parameters will be passed only to the requested states and not to all the target states. Code example:

```
const example = machine({
  A: { add: ["B"] },
  B: {}
});
// method handler
example.A_state = function(param) {
  console.log("A", param);
};
example.B_state = function(param) {
  console.log("B", param);
};
example.add("A", "param");
// prints:
// "A", "param"
// "B", undefined
```

Transition steps

There's six steps to a transition:

1. Lock checks - this includes the queue lock and the machine lock. See the queues section for more info.
2. Relations resolution - computing the active states for the machine after the transition is over.
3. Negotiation handlers - for each of the states which are about to be activated or de-activated. Every of which can say **NO** in which case the transition is cancelled.
4. Set the target states as active
5. Final handlers - for each of the target states which do actual work (allocate and dispose resources, call APIs, etc).
6. Release the machine lock.

Calculating the target states

Requested states combined with active states and the relations between them result in target states of a transition. This phase is **abortable** - if any of the requested states gets rejected, the **transition is aborted**. The only exception to this rule are auto states, which can be accepted separately.

Target states can be accessed using the **transition.to** reference.

```
const example = machine({
  A: { add: ["B"] },
  B: {}
});
example.A_enter = function() {
  this.is(); // -> []
  // this returns target states
  this.transition.to(); // -> ['A', 'B']
```

```
};
// 'A' is a requested state here
example.add("A");
```

Negotiation Handlers

Negotiation handlers (`enter` and `exit`) get called as the third transition step of transition for every state which is going to be activated or de-activated. They are allowed to abort the current transition by returning `false` . Negotiation handlers shouldn't be `async` , as their return has to be `boolean` .

Target states can be accessed using the `this.transition.to()` method, while `this.is()` still returns the base states (before the mutation).

```
const example = machine(["A"]);
example.A_enter = function() {
  return false;
};
example.add("A"); //-> false
example.is(); // -> []
```

Final handlers

Final handlers (`state` and `end`) get called as the fifth transition step. Their purpose is to allocate and dispose resources, call APIs, and perform other actions with side effects.

Just like negotiation handlers, they get called for every target state for every state which is going to be activated or de-activated. Final handlers can be `async` and control the flow cancellation by using abort functions.

Target states can be accessed using the `this.is()` method, while `this.transition.before()` returns the base states (before the mutation).

```
const example = machine({
  A: { add: ["B"] },
  B: {}
});
example.A_enter = function() {
  return false;
};
example.add("A");
```

Abort functions

Because of the way the event loop works theres only two ways to cancel a function / method:

1. Control-flow based (`return` or conditional statements)
2. Generators

AsyncMachine makes the **control-flow based** cancellation fairly easy to use by laveraging a combination of states and their clocks. It exposes the `this.getAbort()` method which then returns a function. Calls to the returned function tell you if you should abort the execution or not. `getAbort` should be called inside of final handlers and can be nested. In case of nesting, inner abort functions influence the outer ones.

Signature:

```
class AM {
  getAbort(states: string[], abort?: () => boolean): () => boolean;
}
```

Example:

```
import delay from "delay";
const example = machine({
  A: {}
});
example.A_state = async function() {
  const abort = this.getAbort(["A"]);
  // wait for 2 seconds
  await delay(2000);
  // at this point `abort()` returns true, as the state's A clock changed
  // from 1 to 2
  if (abort()) return;
  // this line will never get executed
};
// set the state A and then set it again after a second
example.add("A");
setTimout(function() {
  example.set("A");
}, 1000);


// TODO example of nested abort functions
```

Exception As A State

Considering that everything (meaningful) is a state, so should be the fact that an exception was thrown. Every machine has a predefined exception handler, which is the only predefined state.

Signature of the `Exception_state` handler:

```
class AM {
  Exception_state(
    err: Error | TransitionException,
    // eg ['A', 'B']
    target_states: string[],
    // eg ['A']
    base_states: string[],
    // eg 'A_enter'
    exception_src_handler: string,
    // eg ['B']
    async_target_states?: string[]
  );
}
```

Exception During Transition Handlers

Exceptions in AsyncMachine don't break your flow nor stop the machine from running. Instead they add an exception state passing as much information as possible, allowing you to act accordingly. `Exception` and it's `Exception_state` handler are the only predefined state and handler in the base `AsyncMachine` class.

Exception thrown in one of the negotiation handlers:

1. Aborts the whole transition
2. State of the machine stays untouched
3. Add mutation to the `Exception` state is prepended to the queue

Exception thrown in a synchronous final handlers:

1. Transition is during the fourth step, so the machine's state has already been replaced with the target states
2. Not all of the final handlers have been fully executed, so the machine's state isn't integral with the context
3. Add mutation to the `Exception` state is prepended to the queue and the integrity should be manually restored

Exception thrown in an `async` final handlers:

1. Transition is already finished, as async methods are executed on the next tick, so the machine's state has already been replaced with the target states
2. All of the final handlers have finished or returned their `Promis` es, but it's not possible to determine if any of those are stil pending
3. Add mutation to the `Exception` state is prepended to the queue to perform a proper action

Example:

```
const example = machine(["A"]);
example.A_enter = function() {
  throw new Error();
};
example.add("A"); //-> false
example.is(); // -> ['Exception']
```

- exception during a negotiation handler
- exception during a final handler
- exception during a delayed mutation

Custom exception handler

- example
- exception during an exception handler

State relations

There's several types of possible relations between states. Their meaning is very simple, but the mechanism behind resolving final states of a mutation is quite complex. Let's look again at the state definition:

```
interface IState {
  // relations
  add?: string[];
  drop?: string[];
  require?: string[];
  after?: string[];

  // properties
  auto?: boolean;
  multi?: boolean;
}
```

`add` relation

The `add` relation tries to activate the listed states along with itself. Their activation is optional, meaning if any of those won't get accepted, the transition will still go through.

```
const example = machine({
  A: { add: ["B"] },
  B: {}
});
example.add(["A"]); // -> true
example.is(); // -> ['A', 'B']
```

`drop` relation

The `drop` relation prevents from activating or de-activates the listed states. If some of the requested states `drop` other requested states or some of the active states `drop` some of the requested states, the transition will be rejected.

Example of an accepted transition involving a `drop` relation:

```
const example = machine({
  A: {},
  B: { drop: ["A"] }
});
example.add(["A"]); // -> true
example.add(["B"]); // -> true
example.is(); // -> ['B']
```

Example of a rejected transition involving a `drop` relation - some of the requested states `drop` other requested states.

```
const example = machine({
  A: {},
  B: { drop: ["A"] }
});
example.add(["A", "B"]); // -> false
example.is(); // -> []
```

Example of a rejected transition involving a `drop` relation - some of the active states `drop` some of the requested states.

```
const example = machine({
  A: {},
  B: { drop: ["A"] }
});
example.add("B"); // -> true
example.add("A"); // -> false
example.is(); // -> ['B']
```

`require` relation

The `require` relation describes the states required for this one to be activated.

Example of an accepted transition involving a `require` relation:

```
const example = machine({
  A: {},
  B: { require: ["A"] }
});
example.add(["A"]); // -> true
example.add(["B"]); // -> true
example.is(); // -> ['A', 'B']
```

Example of a rejected transition involving a `require` relation:

```
const example = machine({
  A: {},
  B: { require: ["B"] }
});
example.add(["B"]); // -> false
example.is(); // -> []
```

`after` relation

The `after` relation decides about the order of the transition handlers. Handlers from the defined state will be executed **after** the handlers from the listed states.

```
const example = machine({
  A: { after: ["B"] },
  B: {}
});
example.A_state = function() {
```

```
  console.log("A");
};
example.B_state = function() {
  console.log("B");
};
example.add(["A", "B"]); // prints 'B', then 'A'
```

Advanced topics

Now once you have the basic understanding of the way AsyncMachine works we can dive deeper into more advanced subjects. Those are mostly extensions of the things already discussed.

Ways To Abort A Transition

In the previous chapter we discussed the easiest way to abort a transition - negotiation handlers. They are not the only ones though, so lets look at the other ones, each with a working example.

Aborting By Requested States

All the requested states of a mutation have to be accepted, which means not `drop` ped by any of active of about-to-be-active states. There's some ceaviot exceptions from that rule discussed in the Relations Resolution Process section.

```
const example = machine({
  A: { drop: ["B"] },
  B: {}
});
example.add("A"); //-> true
example.add("B"); //-> false
example.is(); // -> ['A']
```

In the second mutation state `B` wasn't accepted, because the currently active state `A` have a `drop` relation with it. `B` being an explicitly requested state and a non-accepted one, aborted the transition (the return of the `add` method was `false`).

An edge case are auto states, which are always requested in a group, but only one need to be accepted to make a transition pass the requested states requirement.

Aborting By Negotiation Handlers

As already shown in the Negotiation Handlers section you can easily abort any transition by returning `false` from either the `enter` or `exit` handlers (to abort a `drop` mutation). What's worth noticing is that **any** of the target states can abort a transition this way, not only handlers belonging to the requsted states.

```
const example = machine({
  A: { add: ["B"] },
  B: {}
});
example.B_enter = function() {
  return false;
};
example.add(["A", "B"]); //-> false
example.is(); // -> []
```

The transition to `A, B` was rejected by the `B_enter` negotiation handler, thus the machine's state wasn't mutated. It didn't matter that the `B` state wasn't an explicitly requested, as [negotiation handlers] are triggered for every state which is going to be activated or de-activated.

Aborting By An Exception

Another possibility to have a transition aborted is when an exception happens in:

- negotiation handlers
- synchronous final handler (`async` ones happen after the transition ends)

```
const example = machine(["A"]);
example.A_enter = function() {
  throw new Error();
};
example.add("A"); //-> false
example.is(); // -> ['Exception']
```

Refer to the Exception During Transition Handlers section for more details and examples.

Aborting By A Pipe

```
// TODO
```

The target object

Defining methods on every new instance (like in all of the examples here) is usually not the best idea, but worry not, theres better ways. One of then is the **target object** and others are discussed in the Object Oriented APIs section.

The `setTarget()` method simply redirects the transition handler calls to an outside object. You can define and instantiete that object in whatever way works for you, so thats the most flexible solution.

Example:

```
const example = machine(["A"]);
class Target {
  A_state() {
    console.log("A_state");
  }
}
const target = new Target();
example.setTarget(target);
example.add("A"); // prints 'A_state'
```

In the example above the `A_state` method is defined on the Target's prototype and thus doesn't get duplicated with every instance. Calling super methods also works well with this solution, although you have to keep in mind that the default `Exception_state` handler is defined on the machine's prototype. Refer to the exception handling section for more details.

Waiting for states sets

While waiting for one specific state is easy by using the event emitter API, waiting for a set of active states isn't trivial. That's one AsyncMachine provides two async methods making it fairly easy:

- `when(states: string[]): Promise<void>`
- `whenNot(states: string[]): Promise<void>`

`when` returns when all of the given states are set, while `whenNot` works the opposite - returns when all of the given state's aren't set. Remember that it's an `async` function, and waiting for it is equivalent of event emitter's `once` method, not `on` - that means the code after `await` will be executed only once.

Example:

```
const example = machine({
  A: {},
  B: {}
});
async function waitTest() {
  await example.when(["A", "B"]);
```

```
  console.log("ok");
}
example.add("A");
example.add("B"); // prints 'ok'
example.is(); // -> ['A', 'B']
```

Pipes - connections between machines

Modeling complex systems using one machine only is practically impossible. Just like relations, pipes between machines are supported on the engine level. You can achieve similar functionality by defining event emitter handlers using delayed mutations, but the built in solution provides some additional features.

```
// TODO - list the features
```

Piping API looks like following:

- `pipeRemove( state: string, machine: AsyncMachine, target_state?: string, flags?: PipeFlags)`
- `pipeAll(machine: AsyncMachine, flags?: PipeFlags)`
- `pipe( states?: (TStates | BaseStates) | (TStates | BaseStates)[], machine?: AsyncMachine, flags?: PipeFlags)`

Example:

```
const m1 = machine(["A"]);
const m2 = machine(["A"]);
m1.pipe("A", m2);
m1.add("A");
m2.is(); // -> ['A']
```

The default piping behavior acts as follows:

- piped state in the target machine reflects the same state in the source machine
- the mutation is not negotiable - this means the target machine can reject the state while the source machine still accepts it
- mutation is scheduled on the target machine queue (more on that in the [queues section])

You can modify the default behavior by passing `PipeFlags`, which are implemented as a binary enum. Only `NEGOTIATION` and `INVERT` can be mixed together though.

Piping, similarly to **event emitter listeners** is executed right away in case the piped state is active (on in-active in case of `INVERT`).

```
export enum PipeFlags {
  NEGOTIATION = 1,
  INVERT = 1 << 2,
  LOCAL_QUEUE = 1 << 3,
  // TODO write tests for this
  NEGOTIATION_BOTH = 1 << 4
}
```

Example of piping with a different name and negotiation: `// TODO example`

Example of inverted piping with a different:

```
import { PipeFlags } from "asyncmachine";
const m1 = machine(["A"]);
const m2 = machine(["Z"]);
m1.add("A");
m1.pipe("A", m2, "Z", PipeFlags.INVERT);
m2.is(); // -> ['Z']
```

Example of piping on the source's machine queue: `// TODO example`

Queue and machine locks

Once you have more than one machine and let them talk to each other you can hit a new problem - the order of transitions across the them. Lets consider the following example:

1. machine 1 is during a transition
2. machine 1 tries to mutate machine 2
3. machine 2 starts a transition
4. machine 2 tries to mutate machine 1

What happens now is that machine 1 can't take make a synchronous mutation, because it's **machine lock** is active. Instead the mutation gets **queued into it's local queue** and will be executed once the transition from the first step (and possibly other queued ones) finish, after which the mutation from the point four will be executed. When you try to mutate a locked machine, the mutation method will return `null`. You can check if a machine is currently locked using the `example.duringTransition()` method.

Example / live demo

```
const m1 = machine(["A", "B"]).id("m1");
const m2 = machine(["Z", "Y"]).id("m2");
m1.A_state = function() {
  m2.add("Z");
};
m2.Z_state = function() {
  m1.add("B");
};
m1.add("A");
```

Log:

```
[add] A
[states] +A
[transition] A_state
[add] Z
[states] +Z
[transition] Z_state
[queue:add] B
[add] B
[states] +B
```

External queues and queue locks

Besides machine locks, there're also queue locks. The purpose of those come from a very handy feature which is **scheduling a mutation of a machine on another machine's queue**. Every mutation method can be scheduled on an external queue **by passing the external machine as the first param**:

Example:

```
const m1 = machine(["A"]);
const m2 = machine(["Z"]);
// activate 'A' in m1 using the m2's queue
m2.add(m1, "A");
m1.is(); // -> ['A']
```

Queue-based race condition

Even though AsyncMachine prevents race conditions by using queues, you can still lock it in the following scenario:

1. machine 1 is during a transition
2. machine 2 tries to mutate machine 1 using it's own queue

3. mutation from machine 2 get's cancelled

Example / live demo

```
// define transitions
const m1 = machine(["A", "B"]).id("m1");
const m2 = machine(["Z", "Y"]).id("m2");
m1.A_state = function() {
  m2.add(m1, "B");
};
// make changes
m1.add("A");
```

This type of race condition should be fixed in the next version of AsyncMachine.

Delayed mutations

All the mutations we talked about so far are **immediate**, which means they'll happen right away and return the result. In many cases you'd like to perform a mutation in:

- Listeners

```
const example = machine(["Clicked"]);
example.Clicked_state = function(event) {
  console.log("clicked");
};
document.addEventListener("click", example.addByListener("Clicked"));
document.dispatchEvent(new Event("click"));
// prints 'clicked'
```

- Callbacks (the error param is handled automatically)

```
import * as fs from "fs";
const example = machine(["FileRead"]);
// notice the lack of the `err` param
example.FileRead_state = function(data) {
  console.log("content", data);
};
fs.readFile("foo", example.addByCallback("FileRead"));
// prints 'content', ...
```

- The next tick

```
const example = machine(["A"]);
example.A_state = function() {
  console.log("A");
};
console.log("before");
// equals to setTimeout(example.addByListener('A'), 0)
example.addNext("A");
console.log("after");
// prints 'before', 'after', 'A'
```

In case of listeners and callbacks, the params passed to them are appended to the transition handler's params. The **node-style callbacks** have the error param removed and passed automatically to the `Exception` state. Every type of mutation provides helper methods for those use cases, while still being compatible with external queues (which overloads the first param).

Delayed `add` mutation methods

```
class AM {
  addByListener(
    states: string[] | string,
    ...params: any[]
  ): (...params: any[]) => void;
```

```
  addByCallback(
    states: string[] | string,
    ...params: any[]
  ): (err?: any, ...params: any[]) => void;
  addNext(states: string[] | string, ...params: any[]): number;
}
```

Delayed `drop` mutation methods

```
class AM {
  dropByListener(
    states: string[] | string,
    ...params: any[]
  ): (...params: any[]) => void;
  dropByCallback(
    states: string[] | string,
    ...params: any[]
  ): (err?: any, ...params: any[]) => void;
  dropNext(states: string[] | string, ...params: any[]): number;
}
```

Delayed `set` mutation methods

```
class AM {
  setByListener(
    states: string[] | string,
    ...params: any[]
  ): (...params: any[]) => void;
  setByCallback(
    states: string[] | string,
    ...params: any[]
  ): (err?: any, ...params: any[]) => void;
  setNext(states: string[] | string, ...params: any[]): number;
}
```

Relation Resolution Process

During relations resolution the engine makes decision which states will become the target states of a transition.

Some common patterns and those to consider:

Cross blocking

When two states are mutually exclusive (eg they represent **an async action and the done state**), they set the `drop` relation against each other:

```
const example = machine({
  A: { drop: ["B"] },
  B: { drop: ["A"] }
});
example.add("A");
example.add("B");
example.is(); // -> 'B'
example.add("A");
example.is(); // -> 'A'
```

Implied and blocked by a dropped one

Lets assume the following situation - two states drop each other and one of them has an `add` relation with a third state, although the other one blocks the third state. In that case the `add` state will still be activated along with the one who's adding it, BUT the lookup will be limited only to adjacent (directly related) states.

Example using `Wet`, `Dry` and `Water` (live version)

```
const example = machine({
  Wet: { require: ["Water"] },
  Dry: { drop: ["Wet"] },
  Water: { add: ["Wet"], drop: ["Dry"] }
});
example.add("A");
example.add("B");
example.is(); // -> 'B'
example.add("A");
example.is(); // -> 'A'
```

Log:

```
[add] Dry
[states] +Dry
[add] Water
[drop] Dry by Water
[add:implied] Wet
[states] +Wet  +Water  -Dry
```

In the next version of AsyncMachine this behavior should be extended to a deeper lookup.

Logging

The simplest way to inspect whats happening when changing the state of a machine is to turn on it's logging. Log entries include the order of events, called handlers, rejected transitions source of target, but not requested states and pretty much everything else. You can also push your own messages to assert the proper time of events.

There's three level of logging:

1. Elementary state changes and important messages
2. Detailed relation resolution, called handlers, queued and rejected mutations
3. Verbose message like postponed mutations, pipe bindings, auto states

Example:

```
// LOG_LEVEL can be 1|2|3
const LOG_LEVEL = 1;
const example = machine({
  A: {},
  B: { auto: true }
});
// logLevel() method is chainable
example.logLevel(LOG_LEVEL).id("example");
// register a handler for state A
example.A_state = function() {};
// register a rejecting handler for B's negotiation handler
example.B_enter = function() {
  return false;
};
example.add("A");
```

Log level 1

```
[example] [states] +A
```

Log level 2

```
[example] [add] A
[example] [states] +A
[example] [transition] A_state
```

```
[example] [transition] B_enter
[example] [cancelled] B, A by the method B_enter
```

Log level 3

```
[example] [add] A
[example] [states] +A
[example] [transition] A_state
[example] [add:auto] state B
[example] [transition] B_enter
[example] [cancelled] B, A by the method B_enter
```

Consumer API overview

Promises, callbacks and emitters

AsyncMachine tries to support all the possible API patterns present in JavaScript like Promises, node callback, event emitter listeners. Below a list of where each of those is present:

Promises

Promises are present in final transition handlers (in form of `async` methods) and indirectly in every delay mutation method, which after getting called exposes a corresponding `Promise` object as the `instance.last_promise` attribute.

Example:

```
import * as fs from "fs";
const example = machine(["FileRead"]);
fs.readFile("foo", example.addByCallback("FileRead"));
example.last_promise.then(data => {
  console.log("content", data);
});
// prints 'content', ...
```

Callbacks (node-style)

Difference between node-style callbacks and event emitter listeners is **the way they handle errors**. Common callback looks like `function(err, data) { /* ... */ }` and the whole node's standard library uses this pattern. AsyncMachine supports it too and **automatically handles the error param**, passing it as an exception mutation (in case of an error).

```
import * as fs from "fs";
const example = machine(["FileRead"]);
// notice the lack of the `err` param
example.FileRead_state = function(data) {
  console.log("content", data);
};
fs.readFile("foo", example.addByCallback("FileRead"));
// prints 'content', ...
```

Event emitter listeners

Event emitter listeners are supported for delayed mutations in the same way as node-style callbacks. You want to use them to work with the DOM API and libraries exposing event emitters.

Event emitter API

Besides supporting event emitters as a mutation input, AsyncMachine exposes it's own event using the underlying emitter. Important difference between a standard one and the one used here is cancellation - once a listener returns `false`, the event propagation stops. It's used mostly by the negotiation transition handlers. The list of all events emitted by an instance of AsyncMachine can be found in the API docs along with the event emitter API.

You probably won't have to work with them, but the most noticeable one is `tick`, which fires every time the machine's state changes, which is usefull with eg performing repaints.

Example:

```
// define transitions
const example = machine(["A", "B"]);
example.add("A");
example.once("tick", (states_before: string[]) => {
  console.log(states_before);
});
example.add("B"); // prints '["A"]'
```

Object Oriented APIs

All of the examples here use the `machine` factory, as it's the shortest way to compose a working machine. This factory is just a facade to the main `AsyncMachine` class (API docs) which you can easily subclass.

While using the OOP interface you have to do the state registration manually. More than that, when using TypeScript, you have to register them for every subclass because of the way attributes get initialized after the TS -> JS transpilation.

If all of the defined properties are states, then you can you the `registerAll()` method, but if you mix it with other data, you have to manually specify state names.

Example:

```
import AsyncMachine from "asyncmachine";
class Machine extends AsyncMachine {
  A: {};
  B: {};

  constructor() {
    super();
    this.states_all; // -> []

    this.registerAll();
    // ...or...
    this.register("A", "B");

    this.states_all; // -> ['A', 'B']
    this.add("A"); // -> true
    this.is(); // -> ['A']
  }
}
```

Prototypal Inheritance

Defining listeners on every instance is wasteful and defining classes can be to expresive for some, that's why AsyncMachine provides **prototypal inheritance** API. This way of inheritance can not only simulate classic OOP class model, but also dynamically assign prototypes to objects. The latter solution will be used here, but first let's illustrate a simple example using plain JS:

```
const a = { x: 1, y: 1 };
const b = Object.create(a);
b.x = 2
console.log(b.x, b.y) // prints '1 2'
console.log(a.x, a.y) // prints '1 1'
```

What happened above, is that the variable `b` inherited all the properties from variable `a`, but every new assignment made on `b` won't propagate to `a`. That's exactly the way **prototypal children** work in AsyncMachine. Consider the following example:

```
const parent = machine(["A", "B"]);
parent.B_enter = function() {
```

```
  console.log('foo')
};
parent.add('A') // -> true
const child = parent.createChild()
child.is() // -> []
child.add('A') // -> true
child.add('B') // prints 'foo'
child.is() // -> ['A', 'B']
parent.is() // -> ['A']
```

Prototypal children inherit:

- transition handlers
- registered states
- class methods
- log handlers

The following get reset:

- active states
- state clocks
- queue
- machine lock
- queue lock
- event listeners
- pipes

TypeScript types generator

AsyncMachine is written in TypeScript as aims to provide full auto-completion while developing your own machines. This includes the states you define and the whole event emitter interface (including transition handler events). There's a dedicated tool for this called `am-types` which ships with the npm module.

```
$ am-types --help

  Usage: am-types <filename> [options]


  Options:

    -o, --output [output]  save the result to a file, default `./<filename>-types.ts`
    -e, --export <name>    use the following export
    -V, --version          output the version number
    -h, --help             output usage information

  Examples:

    $ am-types file.js
    $ am-types file.json
    $ am-types file.js -s
    $ am-types file.js -s -e my_state
```

Debugging

If you got to this section you know everything about AsyncMachine, at least theory, but now new problems arise - complex network of machines with a lot of mutations can be hard to wrap your head around, especially without having proper tooling. Let's start step by step.

There's three ways to debug a machine:

Logging system

Already discussed in the logging section, quick example below:

```
const example = machine(['A']).id('example')
example.logLevel(3)
example.add('A')
```

Outputs:

```
[example] [add] A
[example] [states] +A
```

Dumping the current state

You can dump the current state of a machine using `statesToString(show_inactive: boolean)`

```
const example = machine(['A', 'B', 'C'])
example.add('A')
example.statesToString(true)
```

Outputs:

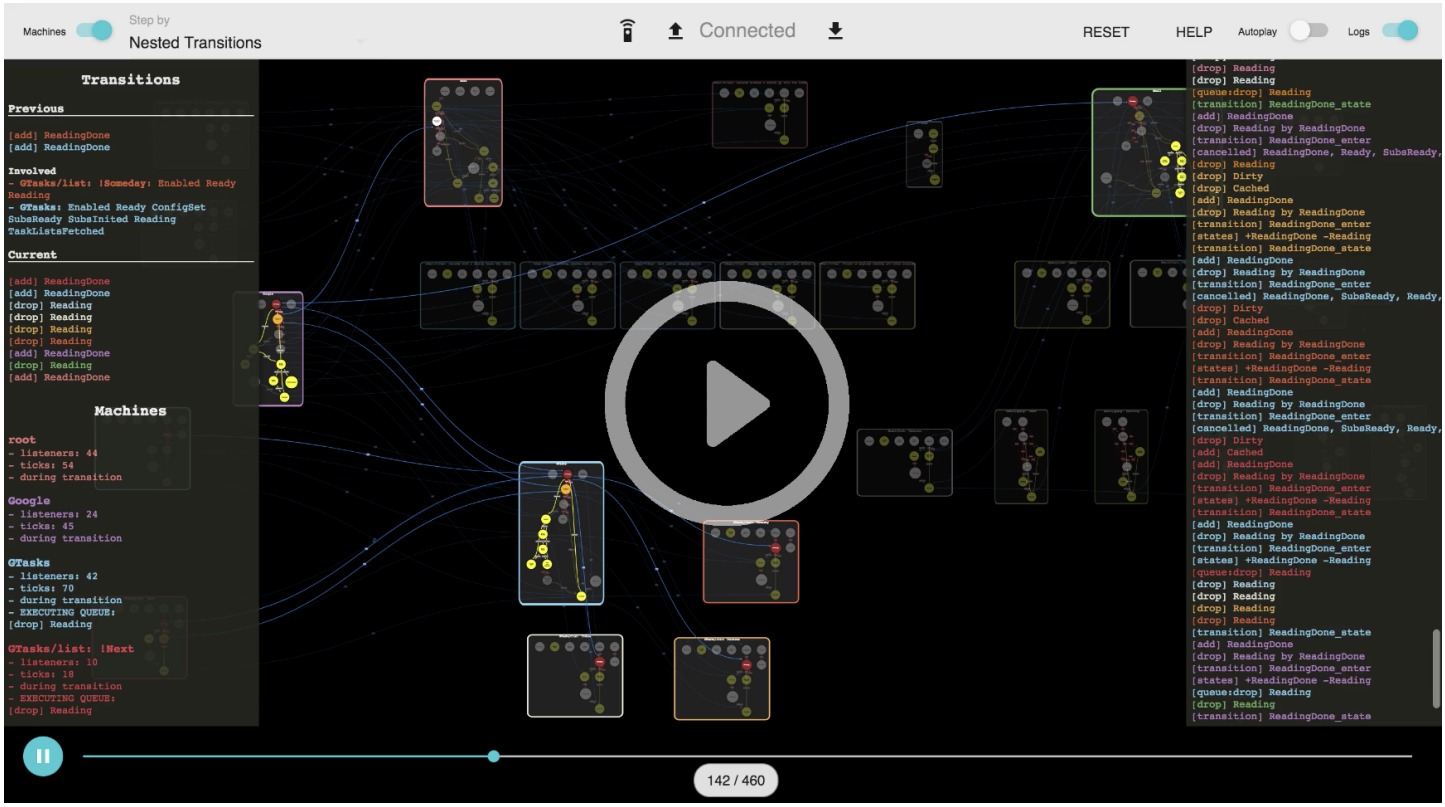
```
example
  A (1)
  -Exception (0)
  -B (0)
  -C (0)
```

AsyncMachine Inspector

AsyncMachine Inspector is both **a debugger and a visual inspector**, which allows stepping through time while seeing:

- synchronized log view
- current
- adjacent transitions
- queues
- custom summary output
- and **even activate states remotely**

It's a great educational tool too and is used by almost every example in this book. It allows you to connect to a live server or load (and save) JSON snapshots.



You can find more about the project on the GitHub:
https://github.com/TobiaszCudnik/asyncmachine-inspector

Or try it live on StackBlitz:
https://stackblitz.com/edit/asyncmachine-inspector-starter

Terminology

- State
- Active state
- In-active state
- Requested states
- Target states
- Mutation
- Transition
- Accepted transition
- Rejected transition
- Negotiation Handlers
- Final Handlers
- Explicit states