**For better understanding go to following link :**

<https://www.youtube.com/watch?v=o3A9EvMspig&list=PLPkE_tfQzWAX6YyTqk81JKl5n5M2wZPa4>

**Redux-Saga**

import { createSagaMiddleware } from 'redux-saga';

const sagaMiddleware = createSagaMiddleware();

const store = createStore(

    reducer,

    applyMiddleware(sagaMiddleware)

);

sagaMiddleware.run(watchFetchDog);

Effects:

Take :

Consumes the action, pauses at the particular line of code and the code resumes when the action is dispatched. It does not create multiple threads.

Properties of action are passed as yielded variables.

Ex:1

let mySaga= function\* () {

console.log("Saga begins!");

const state= yield effects.take("SET\_STATE");

console.info("Got State... ",state);

}

>Undefined

run(mySaga);

>Saga begins!

>{@@redux-saga/TASK: true, id: 1, name: "mySaga", cont: undefined, joiners: Array(0), …}

dispatch({type:"SET\_STATE",value:42})

>Got State... {type: "SET\_STATE", value: 42}

>{type: "SET\_STATE", value: 42}

Ex2:

let process= function\* (){

let i=0;

while(true){

console.log("process Loop! "+ i++);

yield delay(1000);

}

}

>undefined

let saga=function\* () {

yield effects.take("START\_PROCESS",process);

console.log("Saga got to the end");

}

>undefined

run(saga);

>{@@redux-saga/TASK: true, id: 1, name: "saga", cont: undefined, joiners: Array(0), …}

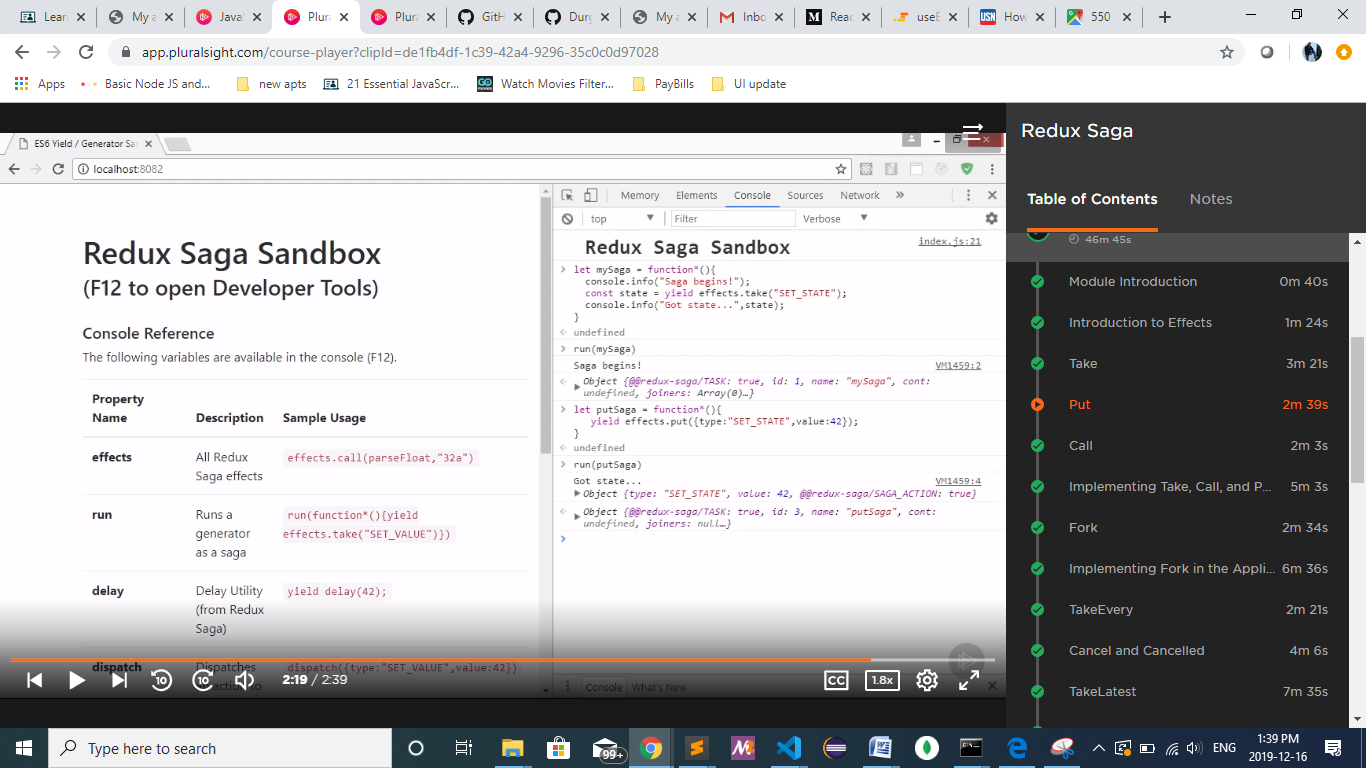
dispatch({type:"START\_PROCESS"})

>Saga got to the end

>{type: "START\_PROCESS"}

Put

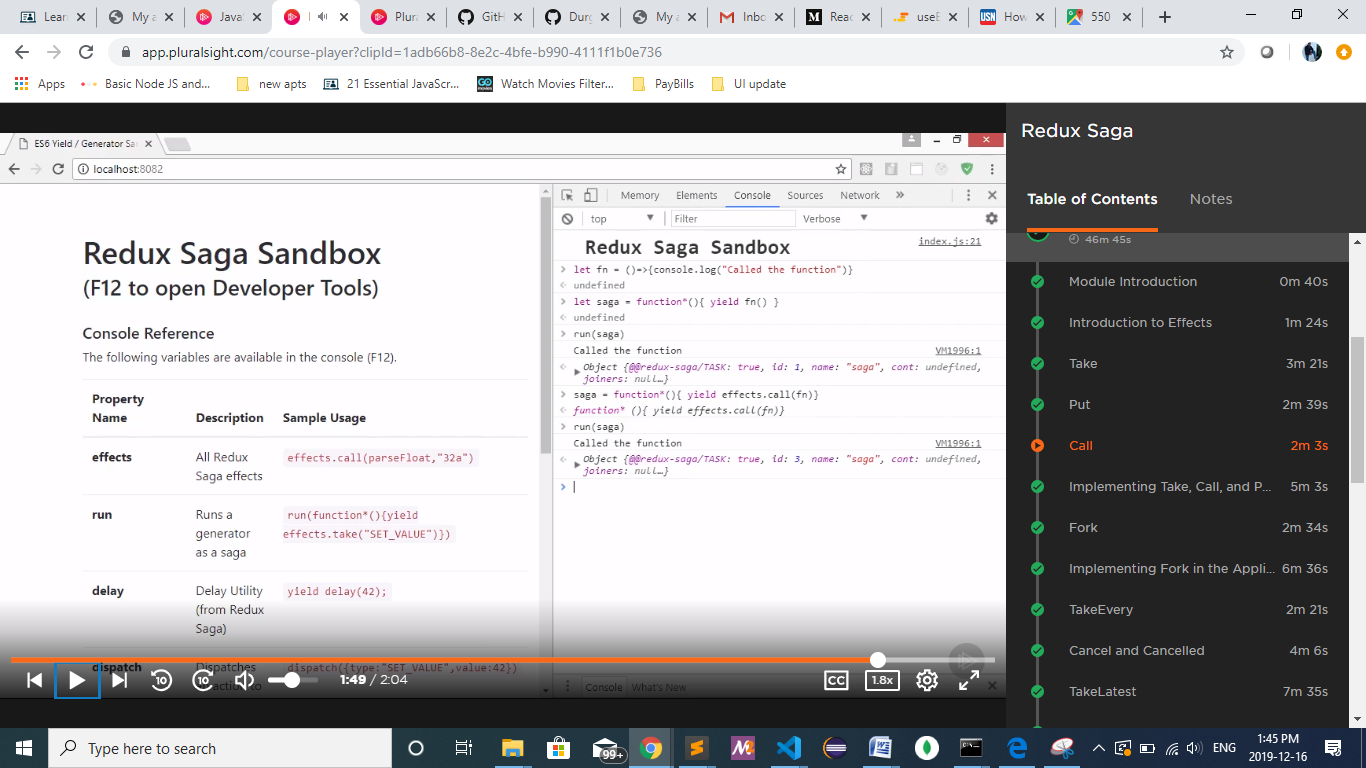
Once take consumes an action , it used to emit or basically dispatches another action inorder to update the state through reducer.



Call

Calls the method (similar to invoking the method , except that it can access the properties of action as parameters.

It is generally used for testing.



Ex of call, take, put, apply:

App.js

import { getStore } from '../getStore';

import React from 'react';

import { Provider } from 'react-redux'

import { MainContainer } from '../components'

import { getCurrentUserInfo } from '../actions'

const store = getStore();

export const App = ()=>(

    <div>

        <Provider store={store}>

            <MainContainer/>

        </Provider>

    </div>

);

store.dispatch(getCurrentUserInfo(`U10000`));

sagas -> currentUserSaga.js

import { take, put, call, apply  } from 'redux-saga/effects'

import fetch from 'isomorphic-fetch';

import {

    GET\_CURRENT\_USER\_INFO,

    setCurrentUser

} from './../actions'

export function\* currentUserSaga() {

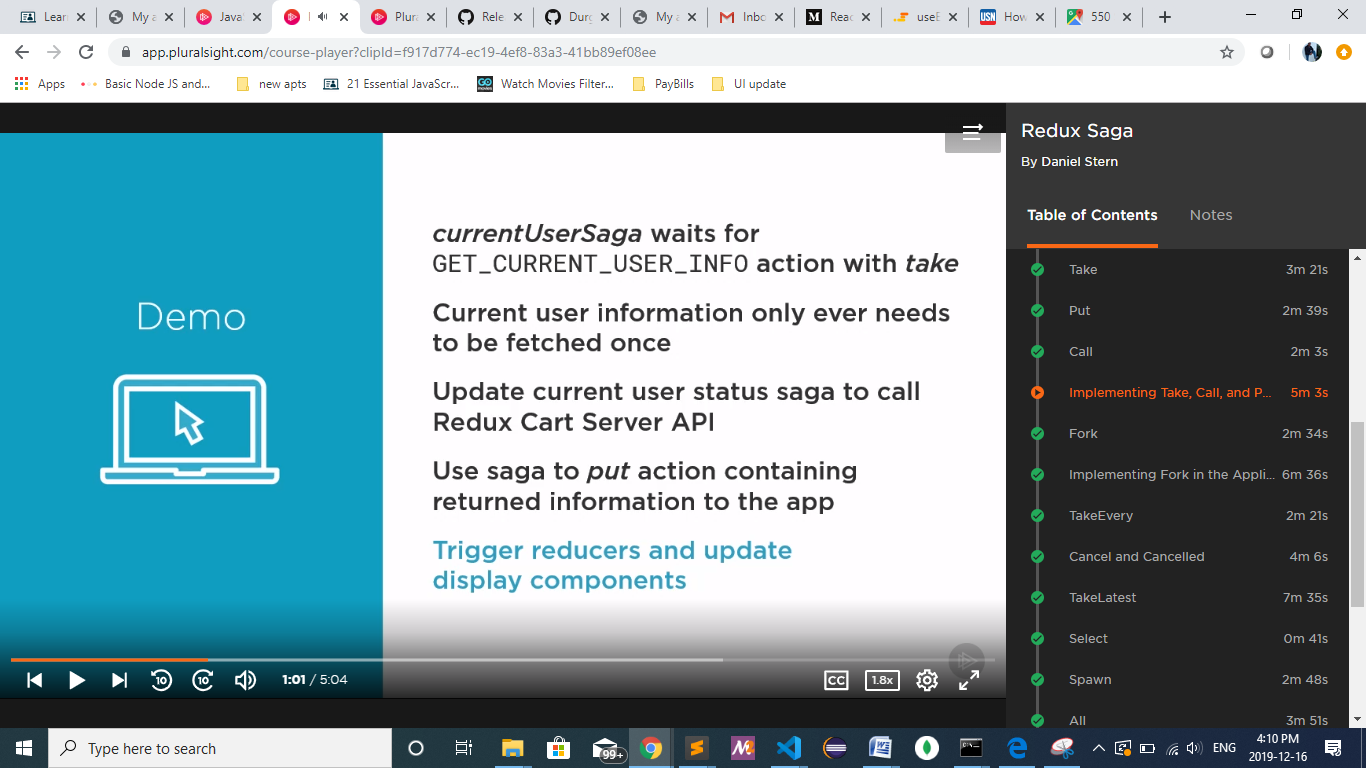
    const { id } = yield take(GET\_CURRENT\_USER\_INFO);

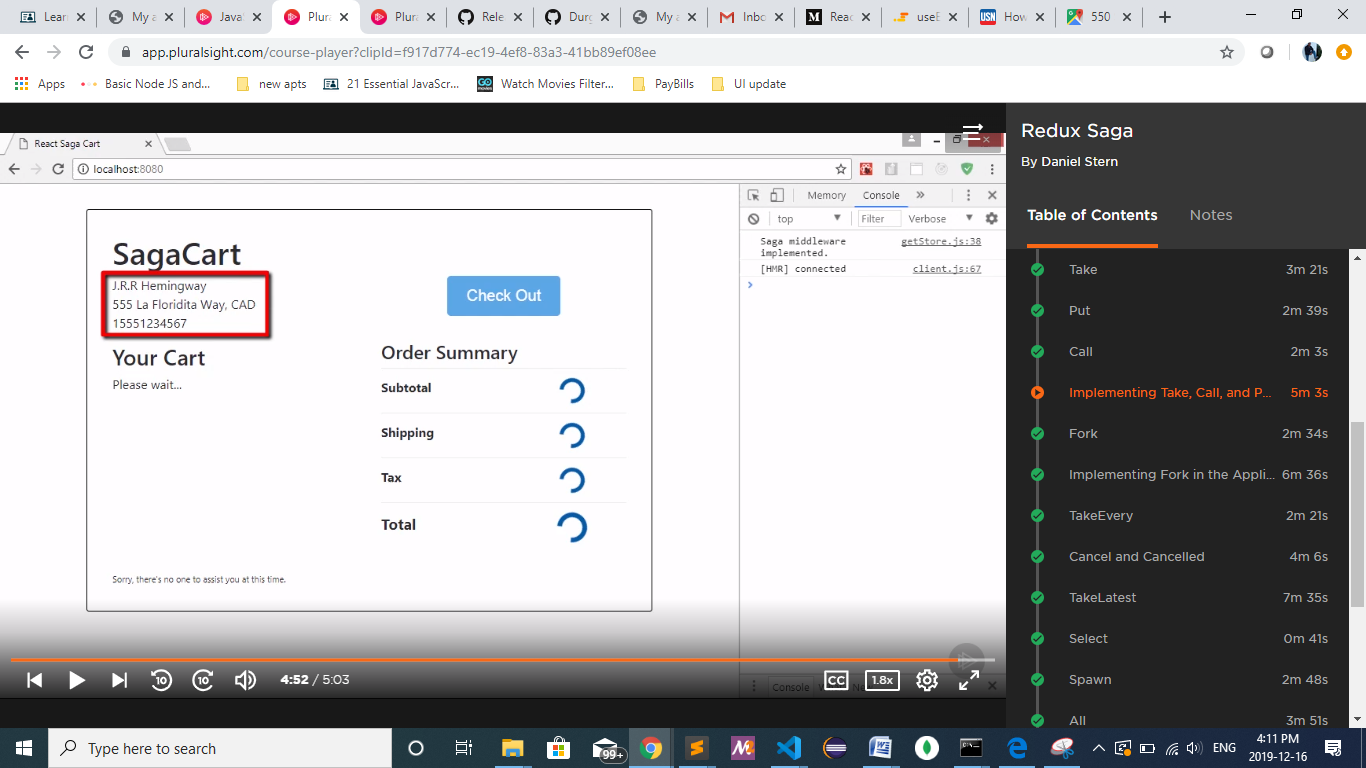
    const response = yield call(fetch,`http://localhost:8081/user/${id}`);

    const data = yield apply(response, response.json);

    yield put(setCurrentUser(data));

}





Apply

**Fork:**

* **It creates multiple thread to handle the process which is invoked multiple times. Every time the process is invoked, our saga will be creating new thread to handle the process.**
* **Cannot access the yielded variable (type and payload of action).**
* **Caller continues to run without passing.**
* **If parent process errors or is cancelled then all the forked processed are cancelled.**
* **They use finally block to show whether error has happened or have been cancelled.**

function\* fn1(){

let i=0;

while(true){

console.log("FN!"+ i++);

yield delay(1000);

}

}

let saga=function\* () {

while(true){

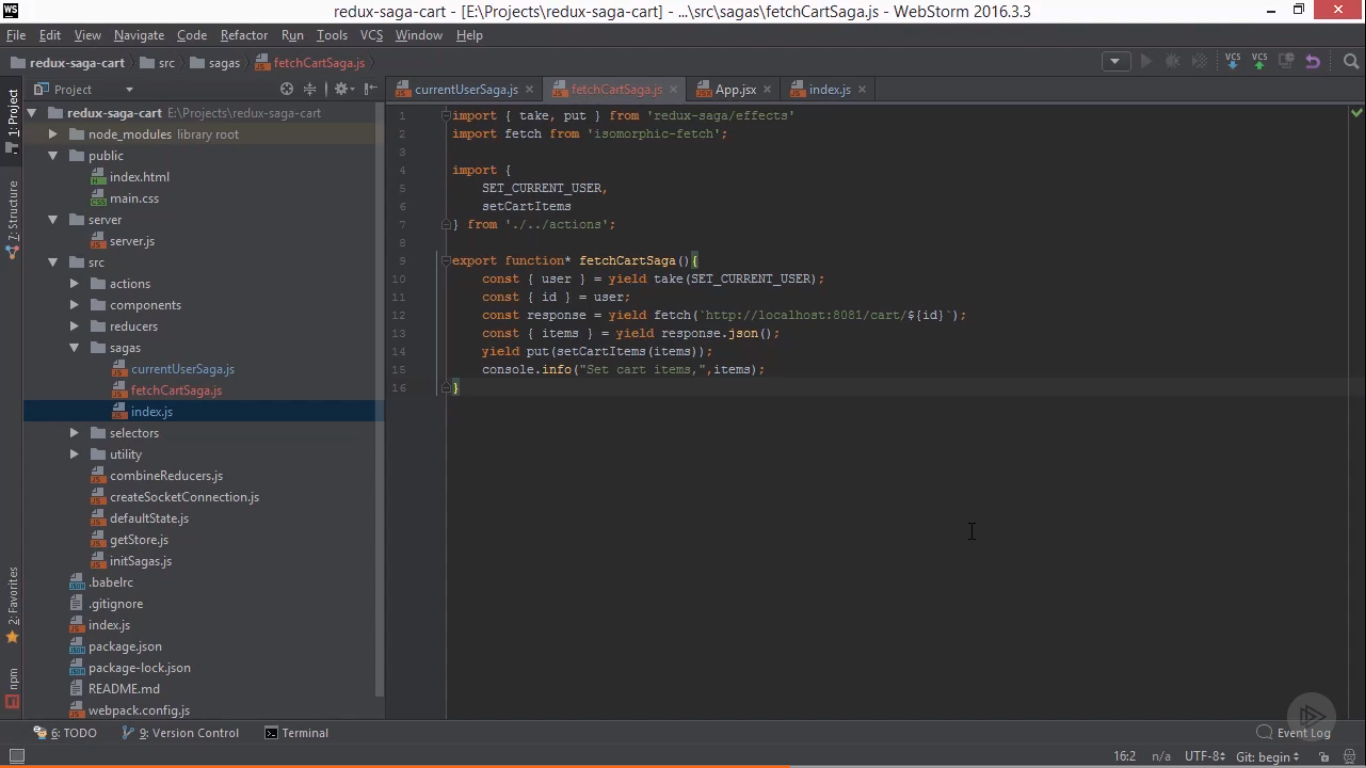
yield effects.fork(fn1);

yield delay(500);

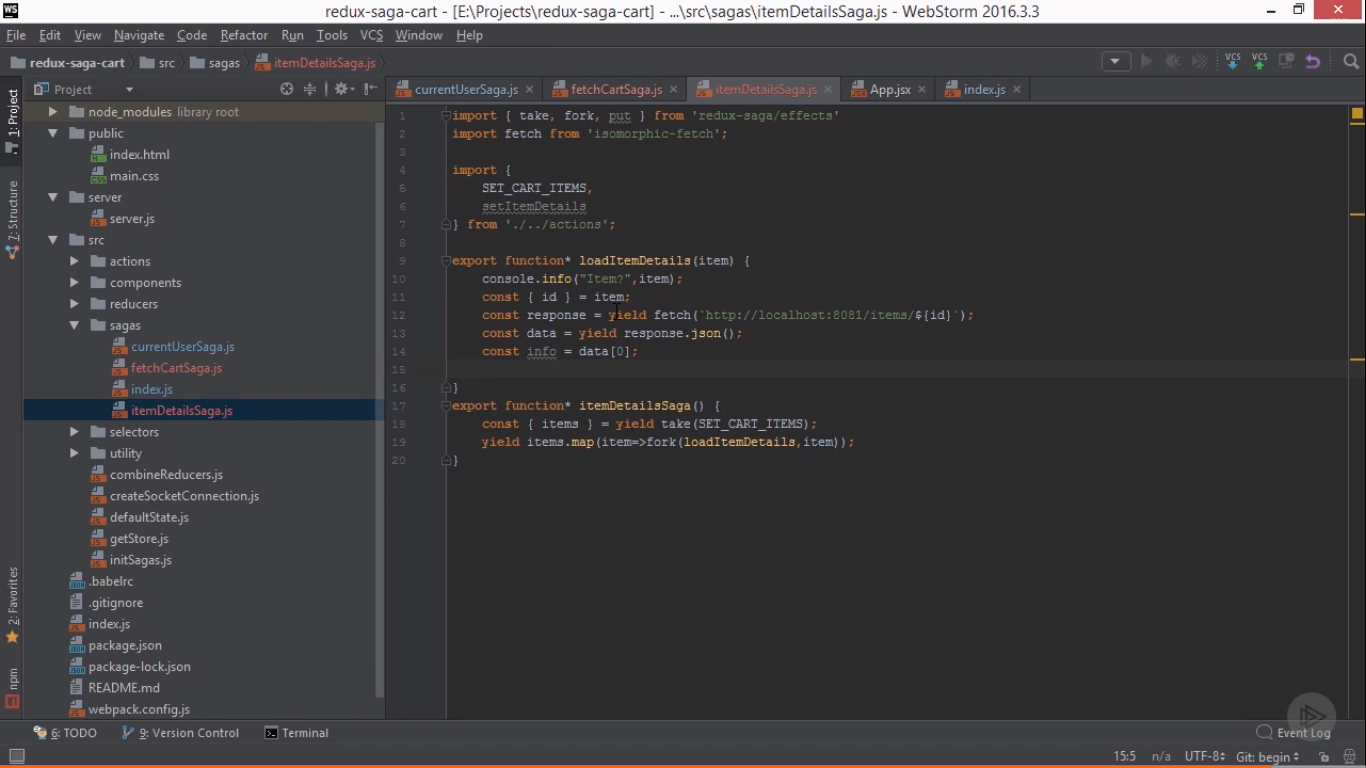
}

}

fetchCartSaga.js



itemDetailsSaga.js



**TakeEvery:**

**It is combination of take and fork effects. It works like take but forks the specified method every time the action is dispatched.**

**The main thread continues to execute unlike take effect.**

**Ex:**

**let process= function\* (){**

**let i=0;**

**while(true){**

**console.log("process Loop! "+ i++);**

**yield delay(1000);**

**}**

**}**

let saga=function\* () {

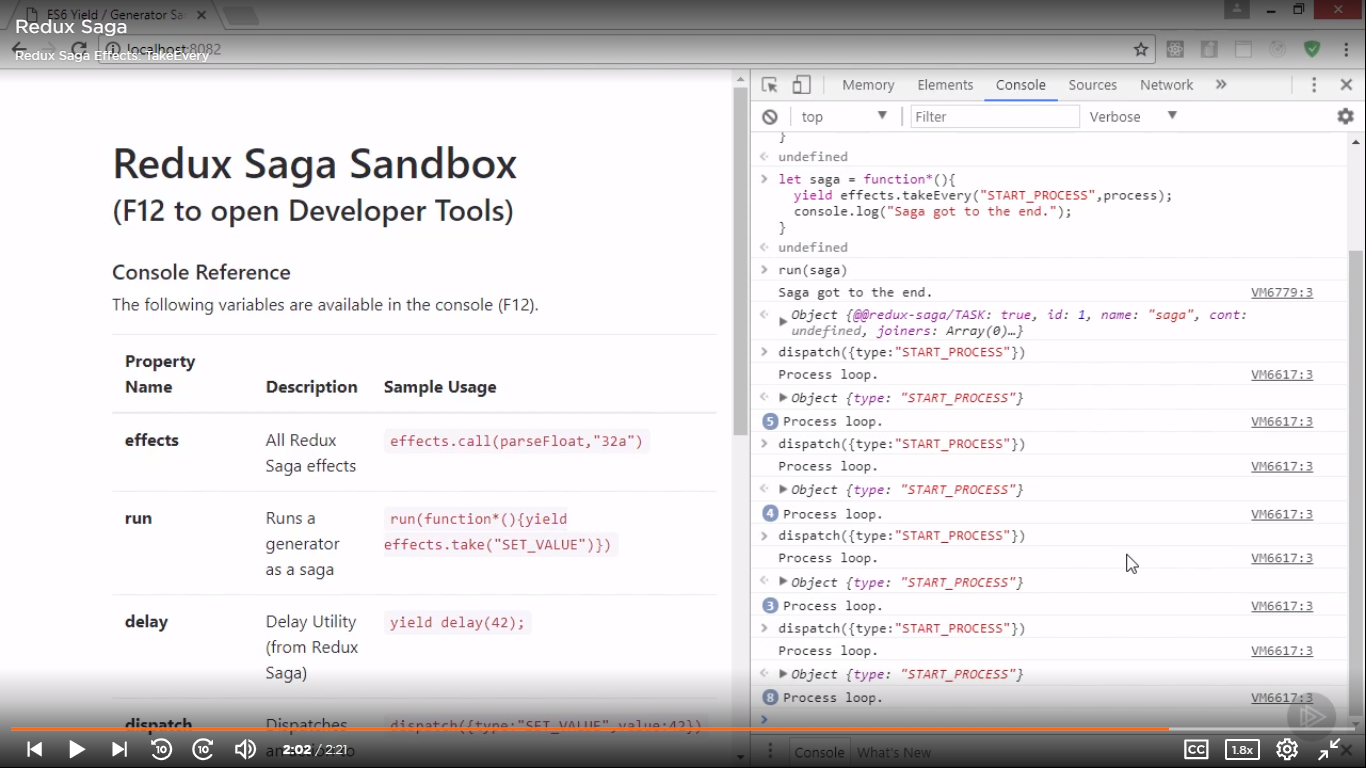
yield effects.takeEvery("START\_PROCESS",process);

console.log("Saga got to the end");

}

run(saga);

VM2803:3 Saga got to the end



Cancel and cancelled:

It stops a forked process at the yield statement.

When a forked process is cancelled, the finally block is invoked.

