Building a Reactive JavaScript System from Scratch

This document demonstrates a highly advanced and informative JavaScript code sample, intended to showcase deep architectural thinking, mastery of the language, and the ability to design scalable reactivity systems like those found in Vue, SolidJS, or MobX.

1. Dependency Tracking Core

```
const bucket = new WeakMap();
let activeEffect = null;

function track(target, key) {
  if (!activeEffect) return;
  let depsMap = bucket.get(target);
  if (!depsMap) {
    bucket.set(target, (depsMap = new Map()));
  }
  let deps = depsMap.get(key);
  if (!deps) {
    depsMap.set(key, (deps = new Set()));
  }
  deps.add(activeEffect);
}

function trigger(target, key) {
  const depsMap = bucket.get(target);
  if (!depsMap) return;
  const effects = depsMap.get(key);
  if (effects) {
    const effectsToRun = new Set();
    effects.forEach(effectFn => {
      if (effectFn !== activeEffect) {
        effectsToRun.add(effectFn);
      }
    });
    effectsToRun.forEach(effectFn => effectFn());
  }
```

```
}
```

2. Reactive Effect

```
function effect(fn) {
  const effectFn = () => {
    cleanup(effectFn);
    activeEffect = effectFn;
    fn();
    activeEffect = null;
  };
  effectFn.deps = [];
  effectFn();
}

function cleanup(effectFn) {
  for (let dep of effectFn.deps) {
    dep.delete(effectFn);
  }
  effectFn.deps.length = 0;
}
```

3. Reactive Handler

```
function reactive(target) {
  return new Proxy(target, {
    get(obj, key) {
      track(obj, key);
      return typeof obj[key] === "object" && obj[key] !== null
        ? reactive(obj[key])
        : obj[key];
    },
    set(obj, key, value) {
      obj[key] = value;
      trigger(obj, key);
      return true;
    },
```

```
  });
}
```

4. Ref Implementation

```
function ref(initialValue) {
  const wrapper = {
    get value() {
      track(wrapper, "value");
      return initialValue;
    },
    set value(newVal) {
      initialValue = newVal;
      trigger(wrapper, "value");
    },
  };
  return wrapper;
}
```

5. Computed Properties

```
function computed(getter) {
  let value;
  let dirty = true;

  const runner = effect(() => {
    value = getter();
    dirty = false;
  });

  return {
    get value() {
      if (dirty) {
        runner();
      }
      track(this, "value");
      return value;
```

```
    },
  };
}
```

6. Watcher (Advanced Effect)

```
function watch(source, cb) {
  let oldVal;
  const runner = () => {
    const newVal = source();
    cb(newVal, oldVal);
    oldVal = newVal;
  };
  oldVal = source();
  effect(runner);
}
```

7. Batched Updates with Microtasks

```
const jobQueue = new Set();
let isFlushing = false;

function flushJob() {
  if (isFlushing) return;
  isFlushing = true;
  Promise.resolve().then(() => {
    jobQueue.forEach(job => job());
    isFlushing = false;
  });
}

function scheduleJob(job) {
  jobQueue.add(job);
  flushJob();
}
```

8. Example Usage

```
const state = reactive({
  count: 0,
  nested: {
    value: 42,
  },
});

effect(() => {
  console.log("The count is:", state.count);
});

state.count++; // Reactive trigger

const r = ref(10);
effect(() => {
  console.log("Ref is", r.value);
});
r.value++;

const doubled = computed(() => r.value * 2);
console.log("Doubled:", doubled.value);

watch(() => state.nested.value, (newVal, oldVal) => {
  console.log(`nested.value changed from ${oldVal} to ${newVal}`);
});
state.nested.value = 100;
```