SolidJS

Fine-grained reactivity with signals and real DOM.

01 Create & render

```
npm init solid@latest   # create-solid
cd solid-project && npm install
npm run dev
```

```
import { render } from "solid-js/web"
render(() => <App />,
  document.getElementById("app")!)
```

The CLI asks about SolidStart, a template and TypeScript. render needs a function and returns dispose.

02 Components run once

```
const Counter = () => {  // body runs once
  const [count, setCount] = createSignal(0)
  const inc = () => setCount((c) => c + 1)
  return (
    <button onClick={inc}>{count()}</button>
  )
}
```

The body is setup code: only JSX that reads a signal updates, as real DOM. Use class, not className.

03 Signals

```
import { createSignal } from "solid-js"
const [count, setCount] = createSignal(0)
count()                    // 0: call it
setCount(5)                // set a value
setCount((n) => n + 1)     // from previous
const [obj, setObj] = createSignal(data,
  { equals: false })       // always notify
```

A signal is a getter: read count(), not count. The call is what subscribes JSX, effects and memos.

04 Derived signals & memos

```
const doubled = () => count() * 2
const total = createMemo(() =>
  list().reduce((sum, n) => sum + n, 0))
total()  // cached until list() changes
```

A derived signal recomputes on every read. createMemo runs once when created, then only when a dependency changes.

05 createEffect

```
createEffect(() => {
  console.log(name(), count())
})
createEffect((prev) => {
  const next = count()
  console.log(`from ${prev} to ${next}`)
  return next  // the next run's prev
})
```

Every signal read inside is a dependency. The first run waits until rendering is done; effects never run during SSR.

06 batch, untrack & on

```
batch(() => { setA(1); setB(2) })
createEffect(() => {
  console.log(first(), untrack(last))
})
createEffect(on(count, (value, prev) => {
  console.log(prev, "->", value)
}, { defer: true }))
createEffect(on([first, last],
  ([f, l]) => console.log(f, l)))
```

batch applies several writes as one update. untrack reads without subscribing. defer: true skips the first run.

07 Event handlers

```
<button onClick={() => save()}>Save</button>
<input onInput={(e) =>
  setValue(e.currentTarget.value)} />
<div on:scroll={onScroll}>{list}</div>
<li onClick={[remove, id]}>{name}</li>
```

onClick and onInput are delegated to the document; on:scroll adds a native listener. [remove, id] calls remove(id, event). Handlers bind once, not reactively.

08 Show, Switch & Dynamic

```
<Show when={user()} fallback={<Login />}>
  <Dashboard />
</Show>
<Switch fallback={<p>Not found</p>}>
  <Match when={isAdmin()}><Admin /></Match>
  <Match when={user()}><Profile /></Match>
</Switch>
<Dynamic component={views[view()]} />
```

Show renders children while when is truthy. Switch renders the first truthy Match, else its fallback. Dynamic (solid-js/web) picks a component or tag at runtime.

09 For & Index

```
<For each={users()} fallback={<p>None</p>}>
  {(user, i) => <li>{i()}: {user.name}</li>}
</For>
<Index each={tags()}>
  {(tag, i) => <span>#{i} {tag()}</span>}
</Index>
```

For keys rows by item: item is a value, index a signal. Index keys by position: item is a signal, index a number.

SolidJS

Props, lifecycle, stores and async data.

10 Props: never destructure

```
// const { label } = props  // reads once
const Field = (props: FieldProps) => {
  const [own, rest] = splitProps(props,
    ["label", "error"])
  const p = mergeProps({ size: 20 }, rest)
  return <input {...p} title={own.label} />
}
```

Props are lazy getters: read props.label where it is used. splitProps returns [picked, rest]; mergeProps sets defaults.

11 children helper

```
const List = (props: ParentProps) => {
  const c = children(() => props.children)
  createEffect(() =>
    console.log(c.toArray().length))
  return <ul>{c()}</ul>
}
```

children() resolves and memoizes children so you can inspect them; otherwise render props.children.

12 Context

```
const ThemeCtx = createContext("light")
const App = () => (
  <ThemeCtx.Provider value="dark">
    <Toolbar />
  </ThemeCtx.Provider>
)
const theme = useContext(ThemeCtx)
```

Call useContext in any child; with no provider it returns the default. createContext<T>() gives T | undefined: wrap it in a hook that throws.

13 onMount & onCleanup

```
const Timer = () => {
  const id = setInterval(tick, 1000)
  onMount(() => console.log("mounted"))
  onCleanup(() => clearInterval(id))
  return <p>{secs()}s</p>
}
```

onMount runs once after the first render. onCleanup runs on unmount, and inside an effect before each re-run.

14 Refs

```
const Form = () => {
  let input!: HTMLInputElement
  onMount(() => input.focus())
  return <input ref={input} />
}
<canvas ref={(el) => draw(el)} />
```

A ref is set at creation, before the element is in the DOM: use it in onMount. A callback ref gets it at once; forward as a prop.

15 Stores

```
import { createStore } from "solid-js/store"
const [state, setState] = createStore({
  filter: "all", todos: initialTodos })
setState("filter", "active")
setState("todos", 0, "done", true)
setState("todos", (t) => t.id === 2,
  "text", (s) => s + "!")
state.todos[0].text    // no () needed
```

A path update reaches only that property's readers. A function or a { from, to } range selects array items. Setting an object shallow merges it.

16 produce & reconcile

```
setState(produce((draft) => {
  draft.users.push(newUser)
  draft.users[1].active = true
}))
setState("users", reconcile(apiUsers))
```

Both from solid-js/store. produce takes mutable-style edits. reconcile diffs by id, so unchanged items keep their DOM.

17 createResource

```
const fetchUser = async (id: string) =>
  (await fetch(`/api/users/${id}`)).json()
const [user, { mutate, refetch }] =
  createResource(userId, fetchUser)
user()        // undefined until loaded
user.loading  // also .error, .state
mutate((u) => ({ ...u, name: "Ann" }))
refetch()     // fetch again
```

Refetches when the source changes and skips while it is false, null or undefined. mutate sets the value without fetching.

18 Suspense, errors & Portal

```
<ErrorBoundary fallback={(err, reset) =>
  <button onClick={reset}>Retry</button>}>
  <Suspense fallback={<p>Loading...</p>}>
    <UserProfile />
  </Suspense>
</ErrorBoundary>
<Portal><Modal /></Portal>
```

Suspense shows its fallback while resources inside load. ErrorBoundary catches render and reactive errors, not event handlers. Portal (solid-js/web) mounts into document.body.