원쌤의 Vue.js 퀵스타트

9. Composition API

1. Composition API란

❖Composition API

- 컴포지션 API는 대규모 Vue 애플리케이션에서의 컴포넌트 로직을 효과적으로 구성하고 재사용할 수 있도록 만든 함수 기반의 API
- Vue3에서 추가되었음
- 이전까지 학습한 내용은 옵션 API(Option API)

❖옵션 API의 불편한 점

- 컴포넌트 내부에 동일한 논리적 관심사 코드가 분리하여 존재함
  - data, methods, computed 등의 옵션별로 분리됨
- 컴포넌트 로직 재사용의 불편함
  - mixin이 존재하지만 불편함
  - 특히 여러개의 mixin이 사용될 때는 사용가능한 속성, 메서드를 확인하기 힘듬

1. Composition API란

❖ 옵션 API 문제점 예시

```
<script>
export default {
    name : "OptionsAPI",
    data() {
        return {
            name: "",
            x: 0, y: 0
        }
    },
    computed : {
        result() {
            return parseInt(this.x, 10) + parseInt(this.y, 10)
        }
    },
    mounted() {
        this.name = "john";
        this.x = 10;
        this.y = 20;
    },
    methods: {
        changeX(strX) {
            let x = parseInt(strX, 10);
            this.x = isNaN(x) ? 0 : x;
        },
        changeY(strY) {
            let y = parseInt(strY, 10);
            this.y = isNaN(y) ? 0 : y;
        },
        changeName(name) {
            this.name = name.trim().length < 2 ? "" : name.trim();
        }
    }
}
</script>
```

name 관련 데이터

calc 관련 데이터

calc 관련 계산된 속성

name 데이터 초기화

calc 데이터 초기화

calc 관련 메서드

name 관련 메서드

1. Composition API란

❖Composition API 예시 1

```
<script>
import { reactive, computed, onMounted } from 'vue';

export default {
    name : "CompositionAPI",
    setup() {
        const nameData = reactive({ name : "" })
        const changeName = (name) => {
            console.log(name)
            nameData.name = name.trim().length < 2 ? "" : name.trim();
        }
        onMounted(()=>nameData.name = "john");


        const calcData = reactive({ x:0, y:0 });
        const result = computed(()=>parseInt(calcData.x, 10) + parseInt(calcData.y, 10));
        onMounted(()=>{
            calcData.x = 10;
            calcData.y = 20;
        })
        const changeX = (strX)=> {
            let x = parseInt(strX, 10);
            calcData.x = isNaN(x) ? 0 : x;
        }
        const changeY = (strY)=> {
            let y = parseInt(strY, 10);
            calcData.y = isNaN(y) ? 0 : y;
        }

        return { nameData, changeName, calcData, result, changeX, changeY }
    }
}
</script>
```

name과 관련된 데이터. 메서드. 생명주기 메서드

calc와 관련된 데이터. 메서드. 계산된 속성. 생명주기 메서드

1. Composition API란

❖Composition API 예시 2 : 재사용가능한 함수로 분리

```
<script>
import { reactive, computed, onMounted } from 'vue';

const useCalc = (x=0, y=0) => {
    const calcData = reactive({ x:0, y:0 });
    onMounted(()=>{
        calcData.x = x;
        calcData.y = y;
    })
    const result = computed(()=>parseInt(calcData.x, 10) + parseInt(calcData.y, 10));
    const changeX = (strX)=> {
        let x = parseInt(strX, 10);
        calcData.x = isNaN(x) ? 0 : x;
    }
    const changeY = (strY)=> {
        let y = parseInt(strY, 10);
        calcData.y = isNaN(y) ? 0 : y;
    }
    return { calcData, result, changeX, changeY };
}

const useName = (name="john") => {
    const nameData = reactive({ name })
    const changeName = (name) => {
        console.log(name)
        nameData.name = name.trim().length < 2 ? "" : name.trim();
    }
    return { nameData, changeName }
}

export default {
    name : "CompositionAPI",
    setup() {
        const nameObj = useName("smith");
        const calcObj = useCalc(100,200);

        return { ...nameObj, ...calcObj }
    }
}
</script>
```

Composition API는 기존 옵션 API의 컴포넌트 구조를 개선하고 컴포넌트 로직의 재사용성을 높일 수 있도록 설계

2. setup 메서드를 이용한 초기화

❖setup() 메서드

- 옵션 API의 data, methods, computed 옵션 대신에 초기화 작업을 수행하는 메서드
- 주요 기능
  - 컴포넌트 상태 초기화
  - 생명주기 메서드 기능 수행
    - beforeCreate, created 단계에서 실행
  - 기타 생명주기 훅(메서드) 등록
  - 이 메서드에서 리턴한 객체의 속성값은 템플릿에서 이용할 수 있음
- 예시

```
import { ref } from 'vue';
export default {
    name : "Calc",
    setup() {
        const x = ref(10);
        const y = ref(20);
        return { x, y }
    }
}
```

2. setup 메서드를 이용한 초기화

❖ 프로젝트 생성

```
npm init vue calc-component-test
cd calc-component-test
npm install
```

- src/components 디렉터리의 모든 파일, 하위 디렉터리 삭제

❖예제 09-01 : src/components/Calc.vue

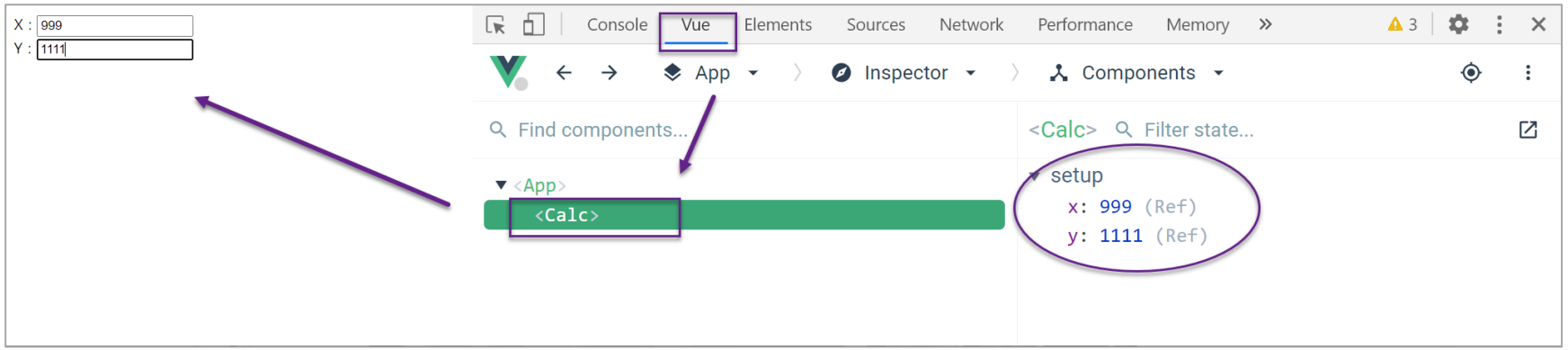
```
<template>
    <div>
        X : <input type="text" v-model.number="x" /><br/>
        Y : <input type="text" v-model.number="y" /><br/>
    </div>
</template>

<script>
import { ref } from 'vue'

export default {
    name : "Calc",
    setup() {
        const x = ref(10);
        const y = ref(20);
        return { x, y }
    }
}
</script>
```

2. setup 메서드를 이용한 초기화

❖예제 09-02 : src/App.vue

- 컴포넌트를 import 하여 사용하는 방법은 기존과 동일함

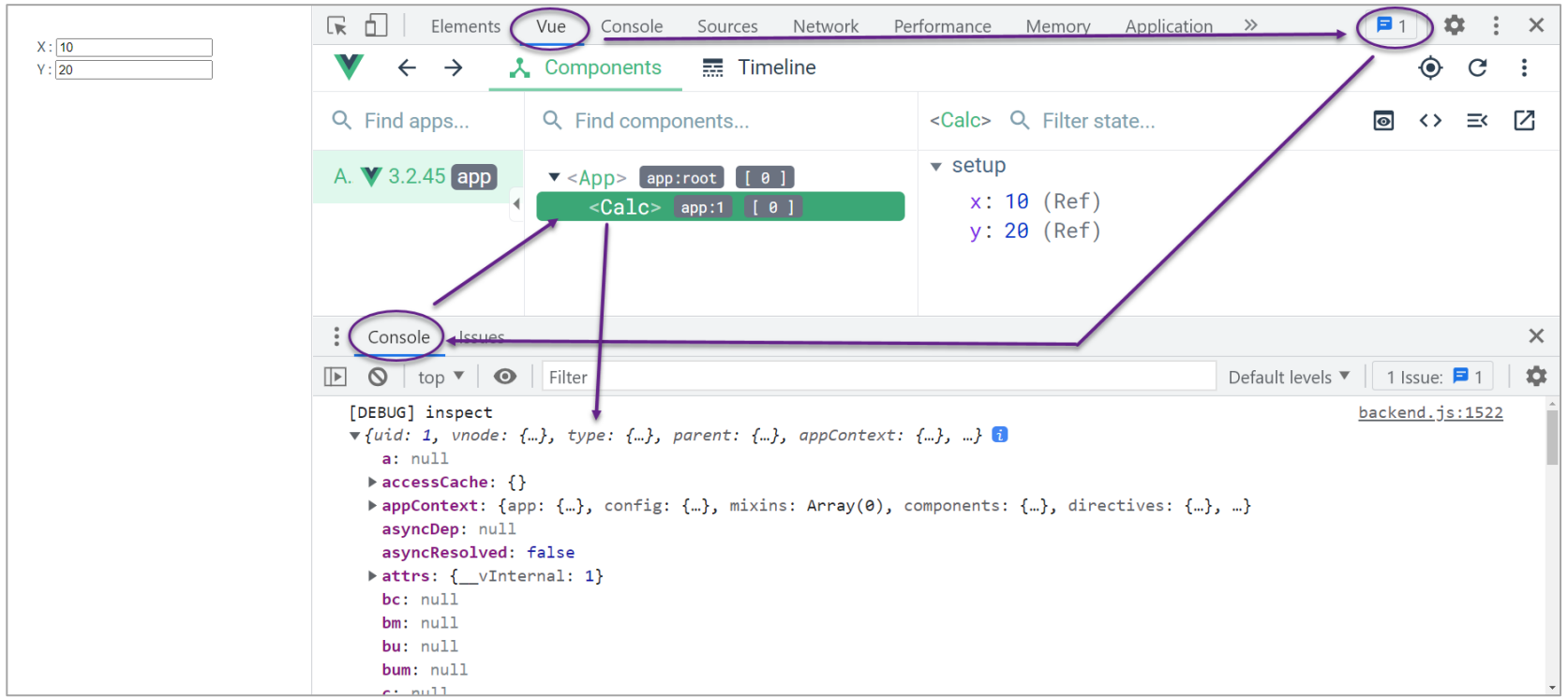
```
1   <template>
2     <div>
3       <Calc />
4     </div>
5   </template>
6
7   <script>
8   import Calc from './components/Calc.vue'
9
10  export default {
11    name : "App",
12    components : { Calc }
13  }
14  </script>
15  <style></style>
```



2. setup 메서드를 이용한 초기화

❖setup() 메서드의 인자

- 첫번째 인자 : props
- 두번째 인자 : context
  - 옵션 API에서 this 를 통해서 제공되던 정보, 기능을 contex가 제공함
  - 예) this.$emit() ----〉 context.emit()
- 개발자 도구에서 context 확인



3. 반응성을 가진 데이터

❖직전 예제에서는 ref()를 사용해서 상태 데이터 생성했음

- 옵션 API에서의 data 옵션과 유사함
- 컴포지션 API의 반응성 데이터 기능 : ref(), reactive()

3.1 ref

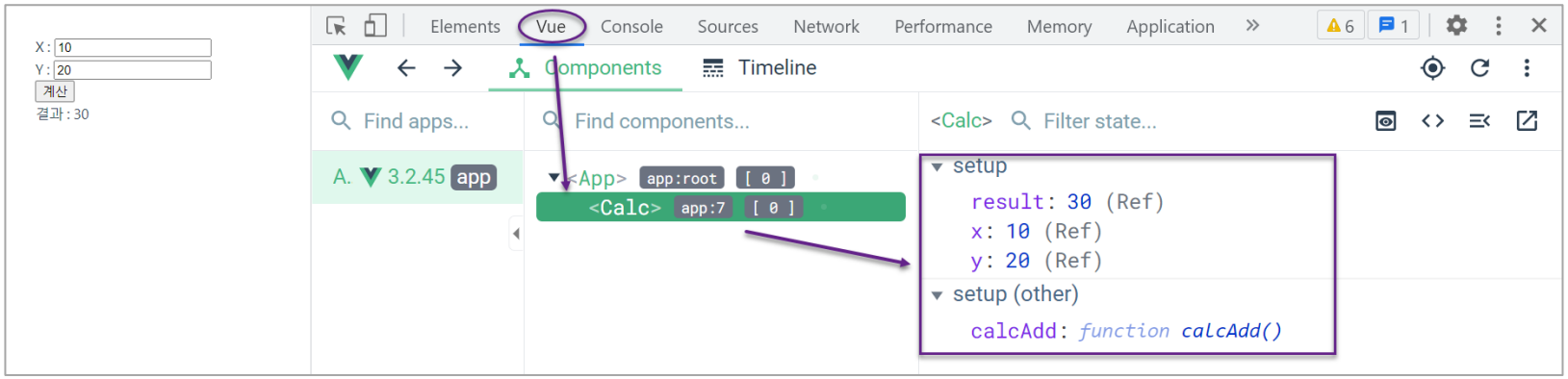
❖ref

- 기본 타입의 값을 이용해 반응성을 가진 참조형 데이터를 생성할 때 사용
- 예제 09-01에서 이미 확인
- ref()의 인자로 초기값을 부여함
- 사용
  - setup() 메서드에서 리턴되어 템플릿에서 이용
  - setup() 메서드 내부에서 정의된 다른 메서드, 계산형 속성 등에서 이용
  - 심지어는 옵션 API의 메서드에서 this. 으로 참조할 수도 있음(권장X)
- 단점
  - 옵션 메서드 내부에서 데이터를 이용할 때는 value 속성으로 접근해야 함

3.1 ref

❖예제 09-03 : src/components/Calc2.vue

```
<template>
  <div>
    X : <input type="text" v-model.number="x" /><br/>
    Y : <input type="text" v-model.number="y" /><br/>
    <button @click="calcAdd">계산</button><br />
    <div>결과 : {{result}}</div>
  </div>
</template>

<script>
import { ref } from 'vue'

export default {
  name : "Calc2",
  setup() {
    const x = ref(10);
    const y = ref(20);
    const result = ref(30);
    const calcAdd = () => {
      result.value = x.value + y.value;
    }

    return { x, y, result, calcAdd }
  }
}
</script>
```



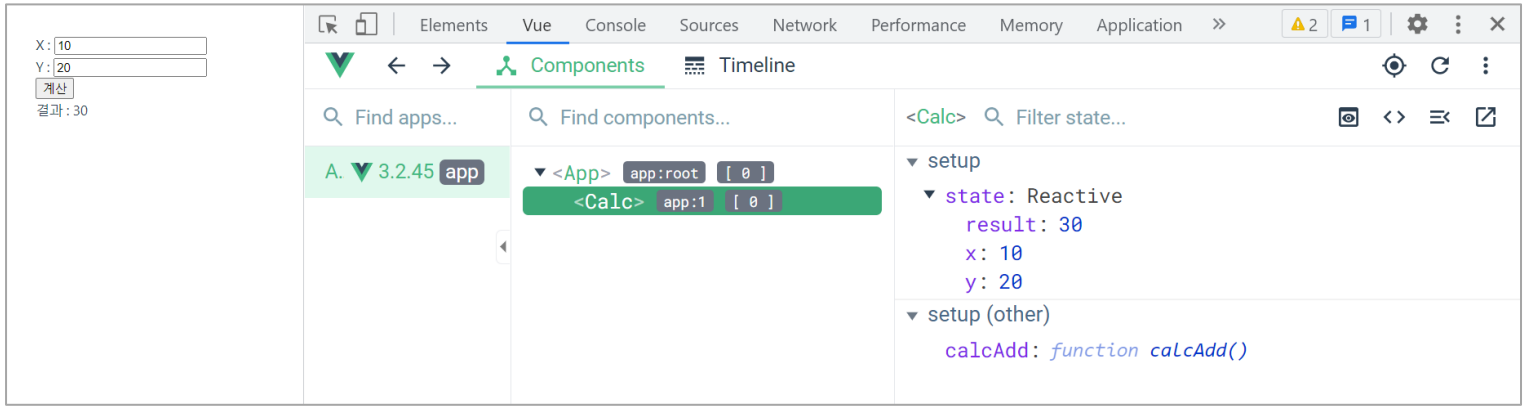
3.2 reactive

❖reactive

- ref와는 달리 객체에 대한 반응성을 가지도록 함

❖예제 09-04 : src/components/Calc3.vue

```
1   <template>
2     <div>
3       X : <input type="text" v-model.number="state.x" /><br/>
4       Y : <input type="text" v-model.number="state.y" /><br/>
5       <button @click="calcAdd">계산</button><br />
6       <div>결과 : {{state.result}}</div>
7     </div>
8   </template>
9
10  <script>
11  import { reactive } from 'vue'
12
13  export default {
14    name : "Calc3",
15    setup() {
16      const state = reactive({ x:10, y:20, result:30 })
17      const calcAdd = () => {
18        state.result = state.x + state.y;
19      }
20      return { state, calcAdd }
21    }
22  }
23  </script>
```



3.2 reactive

❖reactive 주의 사항

- setup() 에서 리턴할 때 reactive 객체 내부의 속성을 리턴하면 반응성을 잃어버림
- 반드시 reactive() 하게 만든 객체를 리턴하여 이용해야 반응성이 유지됨
- 반응성을 잃어버리면 반응성 객체 내부의 값을 변경해도 화면이 갱신되지 않음

4. computed

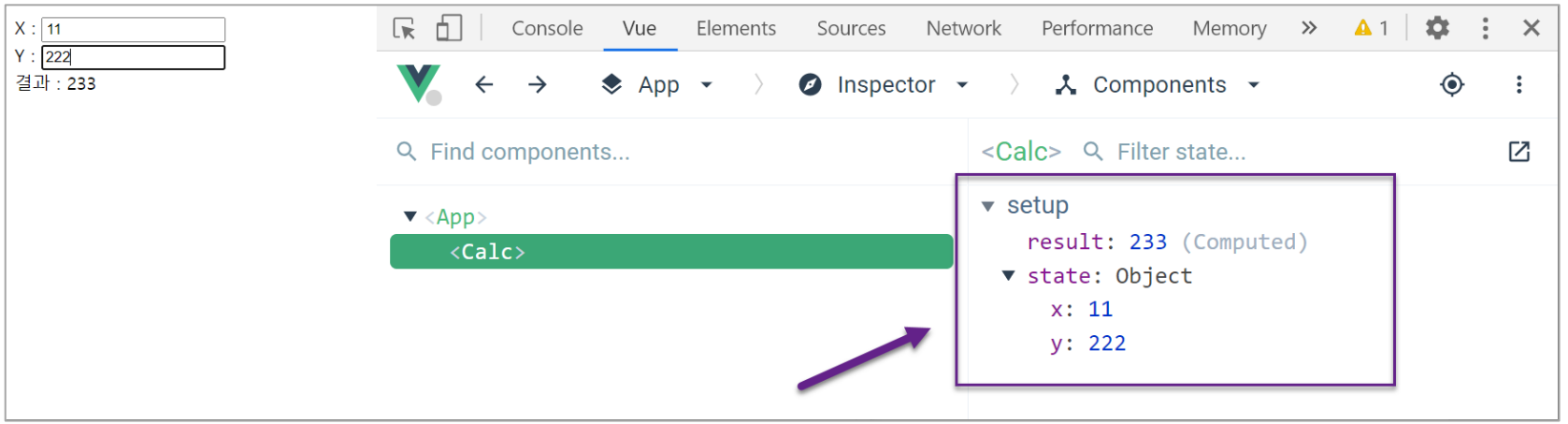
❖computed()

- 옵션 API의 computed 옵션과 유사한 기능을 제공
- 기존 상태, 속성을 이용한 연산 결과에 대한 반응성을 제공함

❖예제 09-05 : src/components/Calc4.vue

- computed()에 의해 생성된 계산된 속성은 템플릿에서는 직접 이용할 수 있지만 〈script〉〈/script〉 내부에서 사용할 때는 반드시 .value 속성을 통해서 접근해야 함.

```
1  <template>
2    <div>
3      X : <input type="text" v-model.number="state.x" /><br/>
4      Y : <input type="text" v-model.number="state.y" /><br/>
5      <div>결과 : {{result}}</div>
6    </div>
7  </template>
8  <script>
9  import { reactive, computed } from 'vue'
10
11 export default {
12   name : "Calc4",
13   setup() {
14     const state = reactive({ x:10, y:20 })
15     const result = computed(()=> {
16       return state.x + state.y;
17     })
18     return { result, state }
19   }
20 }
21 </script>
```



5.1 watch

❖watch

- 옵션 API의 watch 옵션과 동일한 기능을 제공함 --〉 감시자(watcher) 기능

❖ 사용방법

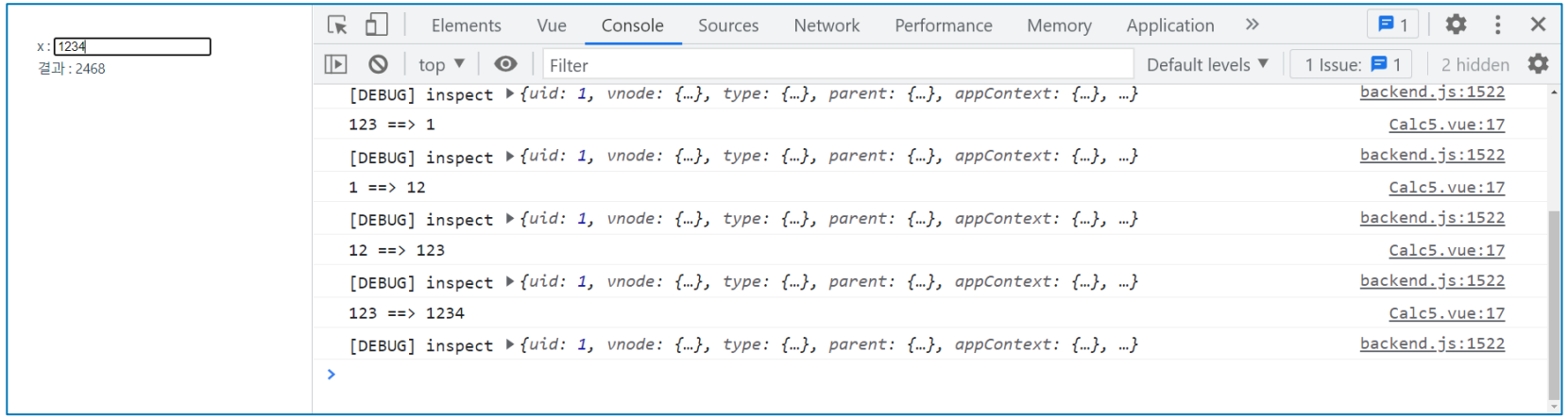
```
watch( data, (current, old) => {
  //처리하려는 연산 로직
})
```

- 첫번째 인자 : 감시 대상 data
- 두번째 인자 : 핸들러 함수
  - 첫번째 인자 : current – 변경된 후의 값
  - 두번째 인자 : old – 변경되기 전의 값
- 만일 감시 대상 data가 ref()로 생성한 반응성 데이터라도....
  - current, old로 접근할 때는 .value 속성 사용하지 않음

5.1 watch

❖예제 09-06 : src/components/Calc5.vue

```
1  <template>
2    <div>
3      x : <input type="text" v-model.number="x" /><br />
4      결과 : {{result}}
5    </div>
6  </template>
7  <script>
8  import { watch, ref } from "vue";
9
10 export default {
11   name : "Calc5",
12   setup() {
13     const x = ref(0);
14     const result = ref(0);
15     watch(x, (current, old) => {
16       console.log(`${old} ==> ${current}`)
17       result.value = current * 2;
18     })
19     return { x, result }
20   }
21 }
22 </script>
```



5.1 watch

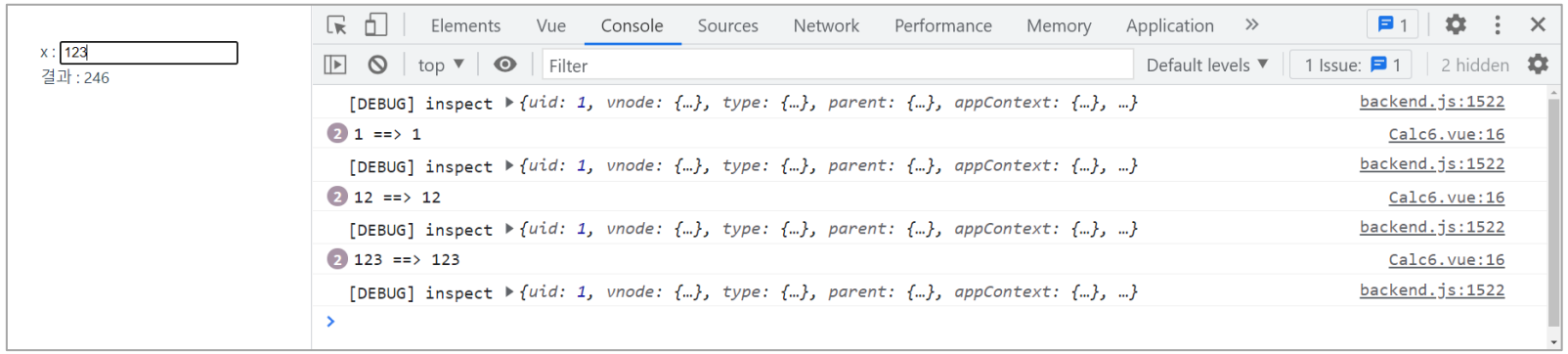
❖reactive()를 이용해 생성한 반응성 객체에 대한 감시자 설정시 주의사항

- 감시대상에 대한 명확한 지정이 필요함
- 명확하지 않으면 핸들러 함수가 불필요하게 여러번 실행될 수 있음. 다음 예제 확인

❖예제 09-07 : src/components/Calc6.vue

```
1  <template>
2    <div>
3      x : <input type="text" v-model.number="state.x" /><br />
4      결과 : {{state.result}}
5    </div>
6  </template>
7  <script>
8  import { watch, reactive } from "vue";
9
10 export default {
11   name : "Calc6",
12   setup() {
13     const state = reactive({ x:0, result:0 });
14     watch(state , (current, old) => {
15       console.log(`${old.x} ==> ${current.x}`)
16       state.result = current.x * 2;
17     })
18     return { state }
19   }
20 }
21 </script>
```

핸들러가 두번 호출됨



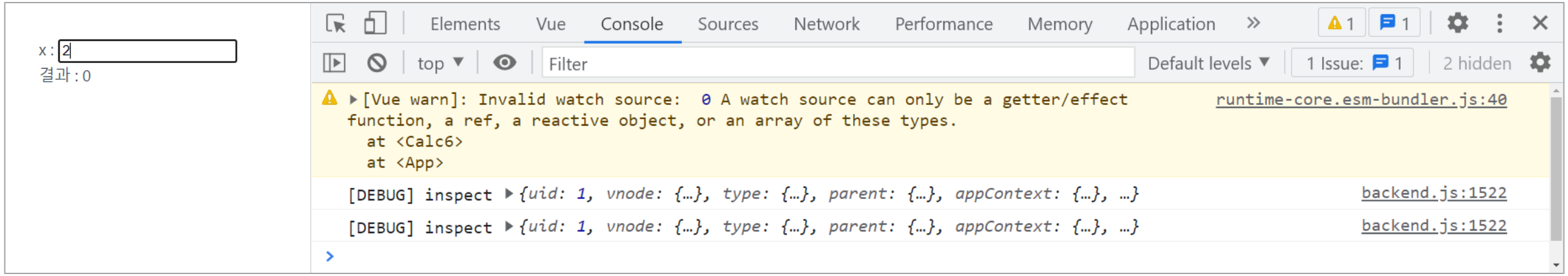
5.1 watch

❖ 예제 09-07을 다음과 같이 변경하면 어떨까?

예제 09-08 : src/components/Calc6.vue의 변경 – 오류발생

```
watch( state.x , (current, old) => {
    console.log(`${old} ==> ${current}`)
    state.result = current * 2;
})
```

▪ 오류 발생



5.1 watch

❖ reactive()로 생성한 반응성 객체 내부의 속성을 감시하려면?

- 함수를 이용해 리턴하면 됨

예제 09-09 : src/components/Calc6.vue의 변경 – 정상 실행

```
watch( ()=>state.x , (current, old) => {
    console.log(`${old} ==> ${current}`)
    state.result = current * 2;
})
```

❖하나의 핸들러로 여러개 감시하고 싶을 때

```
watch( [ a1, a2, ... ] , ( [ currentA1, currentA2, ... ], [ oldA1, oldA2, .... ] ) => {
   ......
})
```

5.1 watch

❖ 여러개 값 감시 예제 : 예제 09-10 - Calc7.vue

```
1   <template>
2     <div>
3       X : <input type="text" v-model.number="x" /><br/>
4       Y : <input type="text" v-model.number="y" /><br/>
5       <div>결과 : {{result}}</div>
6     </div>
7   </template>
8   <script>
9   import { ref, watch } from 'vue'
10
11  export default {
12    name : "Calc7",
13    setup() {
14      const x = ref(10);
15      const y = ref(20);
16      const result = ref(30);
17
18      watch([x, y], ([currentX, currentY], [oldX, oldY] )=>{
19        if (currentX !== oldX) console.log(`X : ${oldX} ==> ${currentX}`)
20        if (currentY !== oldY) console.log(`Y: ${oldY} ==> ${currentY}`)
21        result.value = currentX + currentY;
22      })
23      return { x, y, result }
24    }
25  }
26  </script>
```

5.2 watchEffect

❖Vue3 Composition API에서 반응성 데이터 의존성을 추적하는 새로운 기능

- watch와 유사한 것 같지만 차이가 있음

구분	watch	watchEffect
감시 대상(의존성) 지정	필요함. 지정된 감시 대상 데이터가 변경되면 핸들러 함수가 실행됨	불필요함. 핸들러 함수 내부에서 이용하는 반응성 데이터가 변경되면 함수가 실행됨
변경전 값	이용 가능. 핸들러 함수의 두 번째 인자값을 이용함.	이용 불가. 핸들러 함수의 인자 없음
감시자 설정 후 즉시 실행 여부	즉시 실행되지 않음	즉시 실행

```
watchEffect( () => {
   //반응성 데이터를 사용하는 코드 작성
})
```

5.2 watchEffect

❖예제 09-11 : src/components/Calc8.vue

```
1  <template>
2    <div>
3      X : <input type="text" v-model.number="x" /><br/>
4      Y : <input type="text" v-model.number="y" /><br/>
5      <div>결과 : {{result}}</div>
6    </div>
7  </template>
8  <script>
9  import { ref, watchEffect } from 'vue'
10
11 export default {
12   name : "Calc8",
13   setup() {
14     const x = ref(10);
15     const y = ref(20);
16     const result = ref(0);
17
18     watchEffect(()=>{
19       result.value = x.value + y.value;
20       console.log(`${x.value} + ${y.value} = ${result.value}`)
21     })
22
23     return { x, y, result }
24   }
25 }
26 </script>
```

** Console 확인!!

실행하자마자 watchEffect() 에 등록한 핸들러 함수가 실행되므로 result 가 0이 아닌 30으로 출력됨

5.3 감시자 설정 해제

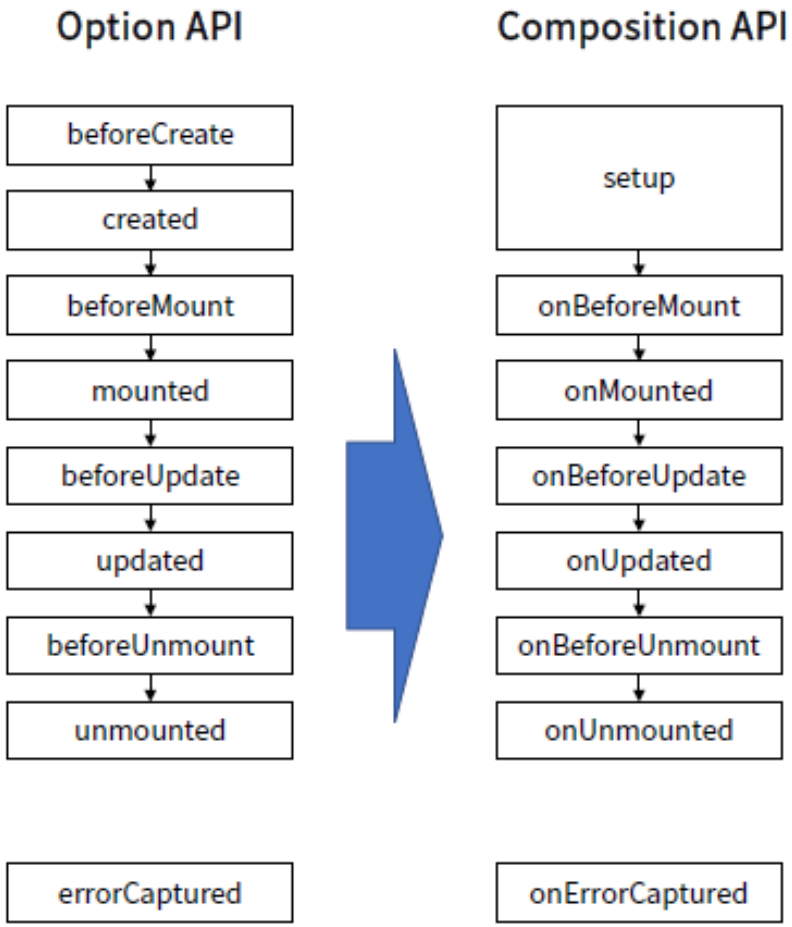
❖ 옵션API에서는...
- watch 옵션을 설정하면 감시자 설정
- 하지만 해제 방법이 없었음

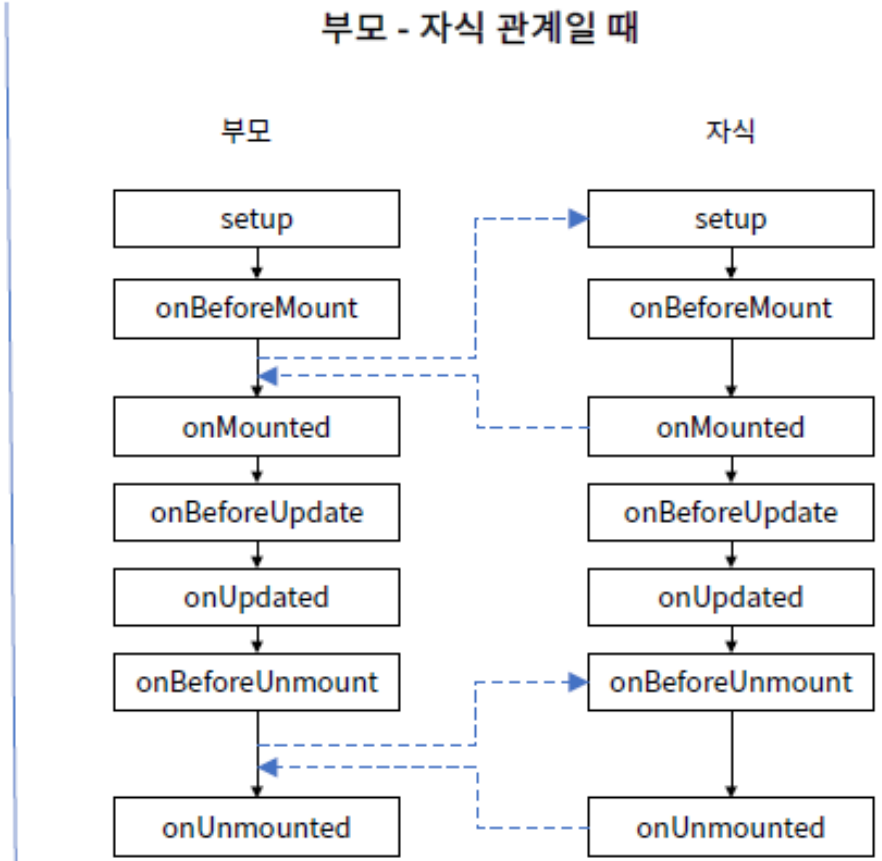
❖ Composition API에서는...
- watch, watchEffect 를 호출한 뒤의 리턴값이 감시자 해제를 위한 핸들러 함수

```
handler = watchEffect( ( ) => { ..... } )
.....
handler()     //설정된 감시자 해제
```

6. 생명주기 훅(Life Cycle Hook)

❖옵션 API의 생명주기 메서드 VS Composition API의 생명주기 메서드(훅)





6. 생명주기 훅(Life Cycle Hook)

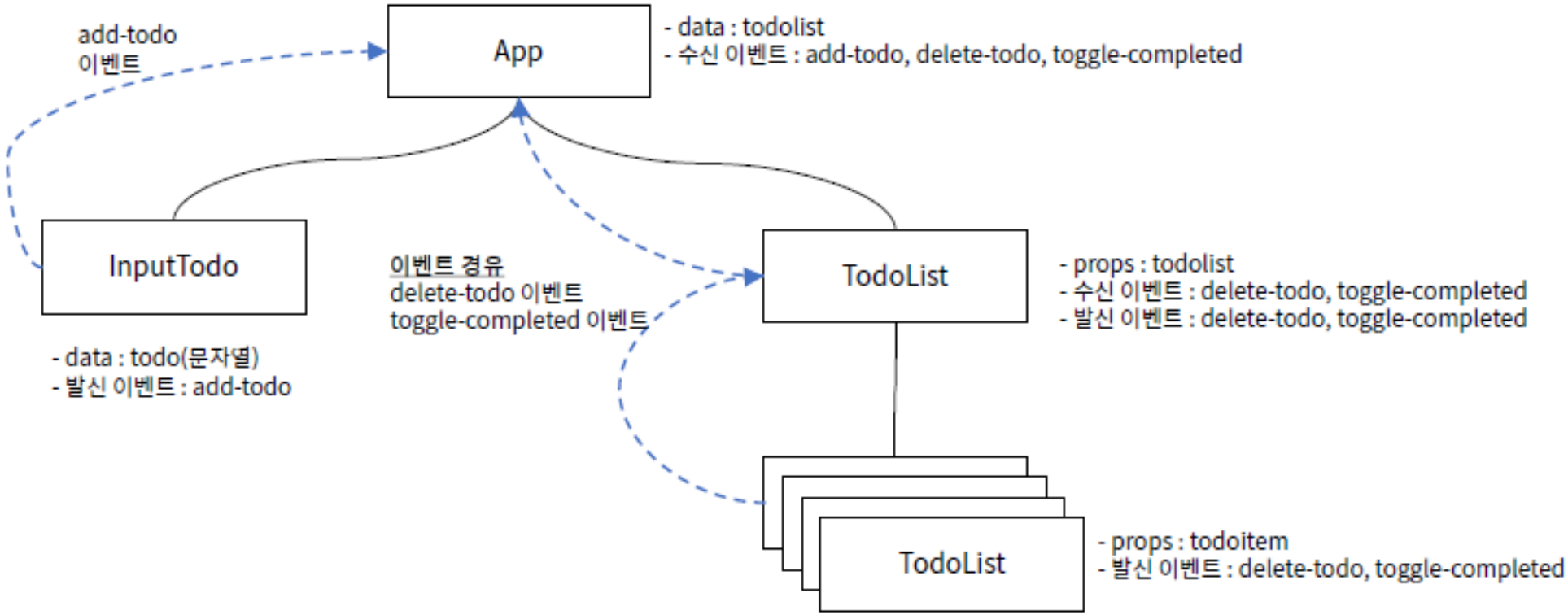
❖ 생명주기 사용방법

```
setup() {
    //onMounted()만을 예시합니다.
    onMounted(()=> {
       ......
    });
    ......
}
```

7. TodoList App 리팩토링

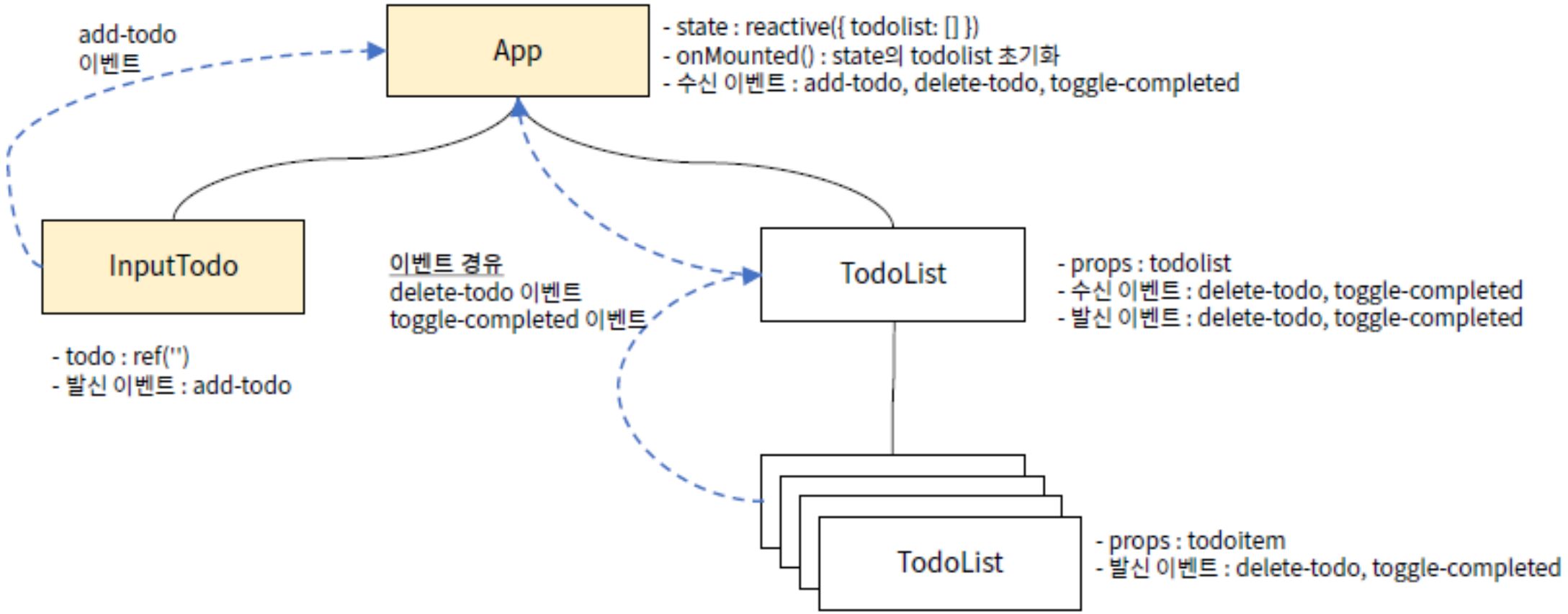
❖ 7.7.2에 옵션 API로 작성한 TodoList App 예제를 Composition API 로 리팩토링
- 이전 예제를 작성하지 않았다면 github 리포의 예제 다운로드
- 7장의 todolist-app772 예제로 시작

❖ 기존 예제의 구조



7. TodoList App 리팩토링

❖리팩토링 예제 구조



7. TodoList App 리팩토링

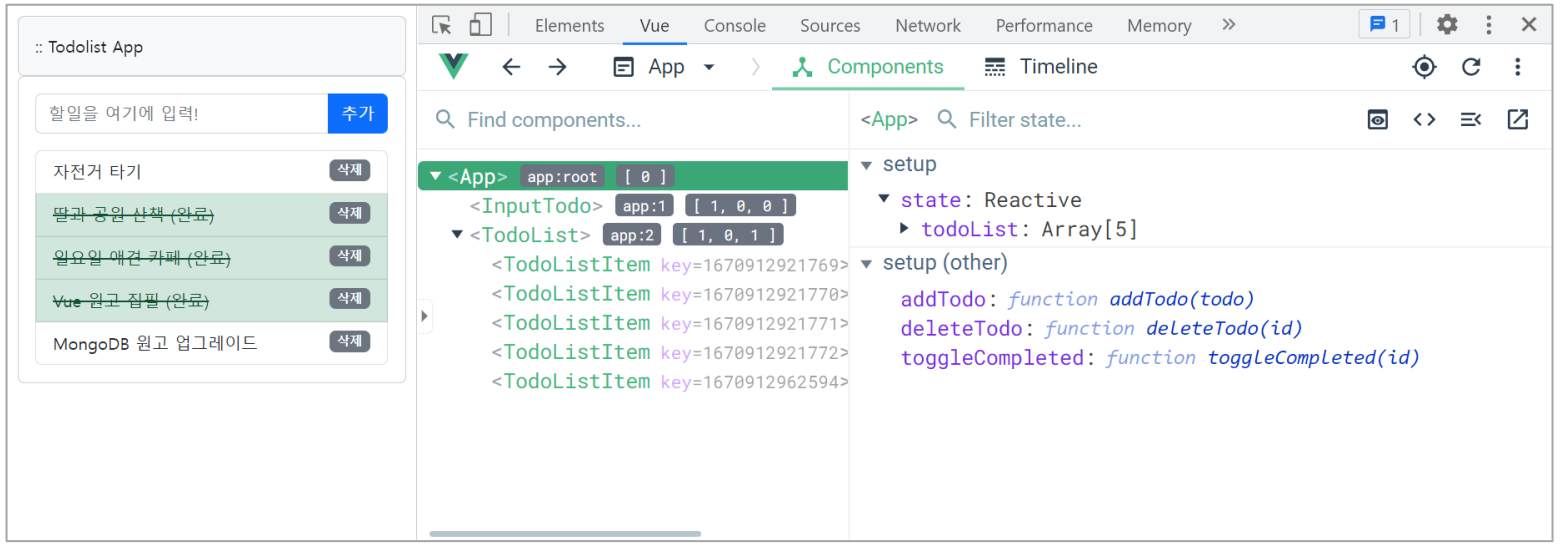
❖예제 09-12 : src/App.vue 새롭게 작성

```vue
<template>
  <div class="container">
    <div class="card card-body bg-light">
      <div classe="title">:: Todolist App</div>
    </div>
    <div class="card card-default card-borderless">
      <div class="card-body">
        <InputTodo @add-todo="addTodo"></InputTodo>
        <TodoList :todoList="state.todoList" @delete-todo="deleteTodo"
          @toggle-completed="toggleCompleted"></TodoList>
      </div>
    </div>
  </div>
</template>

<script>
import { reactive, onMounted } from 'vue'
import InputTodo from './components/InputTodo.vue'
import TodoList from './components/TodoList.vue'

export default {
  name : "App",
  components : { InputTodo, TodoList },
  setup() {
    const ts = new Date().getTime();
    const state = reactive({ todoList : [] })

    onMounted(()=>{
      state.todoList.push({ id: ts, todo:"자전거 타기", completed: false })
      state.todoList.push({ id: ts+1, todo:"딸과 공원 산책", completed: true })
      state.todoList.push({ id: ts+2, todo:"일요일 애견 카페", completed: false })
      state.todoList.push({ id: ts+3, todo:"Vue 원고 집필", completed: false })
    })

    const addTodo = (todo)=> {
      if (todo.length >= 2) {
        state.todoList.push({ id: new Date().getTime(),
          todo:todo, completed: false });
      }
    }
    const deleteTodo = (id) => {
      let index = state.todoList.findIndex((item)=> id === item.id);
      state.todoList.splice(index,1);
    }
    const toggleCompleted = (id) => {
      let index = state.todoList.findIndex((item)=> id === item.id);
      state.todoList[index].completed = !state.todoList[index].completed;
    }

    return { state, addTodo, deleteTodo, toggleCompleted }
  }
}
</script>
```

7. TodoList App 리팩토링

❖예제 09-13 : src/components/InputTodo.vue

```vue
<template> ···
</template>

<script>
import { ref } from 'vue'

export default {
    name : "InputTodo",
    setup(props, context) {
        const todo = ref("");
        const addTodoHandler = () => {
            if (todo.value.length >= 3) {
                context.emit('add-todo', todo.value);
                todo.value = "";
            }
        }
        return { todo, addTodoHandler }
    }
}
</script>
```



8. <script setup> 사용하기

❖〈script setup〉

- 단일 파일 컴포넌트 내부에서 Composition API 사용을 위한 좀 더 편리한 문법적 작성 기능을 제공
- 장점
  - 적은 상용구(boilerplate) 코드 사용으로 간결한 코드를 작성
  - 순수 타입스크립트 언어를 사용해 props, 이벤트 선언
  - 더 좋은 런타임 성능
  - 더 좋은 IDE에서의 타입 추론 성능
- 기존과 차이점
  - 템플릿에서 사용하는 값
    - 최상위에서 선언, 작성된 변수, 함수는 리턴하지 않고도 템플릿에서 이용
  - 컴포넌트 등록
    - components 옵션으로 등록하지 않아도 import 한 컴포넌트를 이용
  - 속성과 발신 이벤트 처리
    - defineProps, defineEmits 함수 이용

```
//기존 방식
setup(props, context) {
   //이벤트를 발신할 때
   context.emit('add-todo', todo)
```

```
// <script setup> 방식
const props = defineProps({
   todoItem : { type : Object, required: true }
})
const emit = defineEmits(['delete-todo','toggle-completed'])
//이벤트를 발신할 때는 다음과 같이
emit('delete-todo', id)
```

8.1 <script setup>이 기존과 다른 점

❖기존과 차이점

- 템플릿에서 사용하는 값
  - 최상위에서 선언, 작성된 변수, 함수는 리턴하지 않고도 템플릿에서 이용
- 컴포넌트 등록
  - components 옵션으로 등록하지 않아도 import 한 컴포넌트를 이용
- 속성과 발신 이벤트 처리
  - defineProps, defineEmits 함수 이용

```
//기존 방식
setup(props, context) {
   //이벤트를 발신할 때
   context.emit('add-todo', todo)
```

```
// <script setup> 방식
const props = defineProps({
   todoItem : { type : Object, required: true }
})
const emit = defineEmits(['delete-todo','toggle-completed'])
//이벤트를 발신할 때는 다음과 같이
emit('delete-todo', id)
```

8.2 TodoList 앱에 <script setup> 적용하기

❖ 예제 09-14 : src/App.vue 변경

```
1  <template> ···
14 </template>
15
16 <script setup>
17 import { reactive, onMounted } from 'vue'
18 import InputTodo from './components/InputTodo.vue'
19 import TodoList from './components/TodoList.vue'
20
21 const ts = new Date().getTime();
22 const state = reactive({ todoList : [] })
23
24 onMounted(()=>{
25   state.todoList.push({ id: ts, todo:"자전거 타기", completed: false })
26   state.todoList.push({ id: ts+1, todo:"딸과 공원 산책", completed: true })
27   state.todoList.push({ id: ts+2, todo:"일요일 애견 카페", completed: false })
28   state.todoList.push({ id: ts+3, todo:"Vue 원고 집필", completed: false })
29 })
30
31 const addTodo = (todo)=> {
32   if (todo.length >= 2) {
33     state.todoList.push({ id: new Date().getTime(),
34       todo:todo, completed: false });
35   }
36 }
37 const deleteTodo = (id) => {
38   let index = state.todoList.findIndex((item)=> id === item.id);
39   state.todoList.splice(index,1);
40 }
41 const toggleCompleted = (id) => {
42   let index = state.todoList.findIndex((item)=> id === item.id);
43   state.todoList[index].completed = !state.todoList[index].completed;
44 }
45 </script>
```

* setup() 사라짐
* setup 내의 마지막 객체 리턴문 사라짐
  -> <script setup> 내부에서 선언된 변수, 함수는 모두 템플릿에서 이용할 수 있음
* components 등록할 필요 없음
  -> import 된 모든 컴포넌트를 템플릿에서 이용할 수 있음

8.2 TodoList 앱에 <script setup> 적용하기

❖ 예제 09-15 : src/components/InputTodo.vue 변경

```
 1 > <template> ···
13   </template>
14
15   <script setup>
16   import { ref } from 'vue'
17
18   const emit = defineEmits(['add-todo'])
19   const todo = ref("");
20   const addTodoHandler = () => {
21       if (todo.value.length >= 3) {
22           emit('add-todo', todo.value);
23           todo.value = "";
24       }
25   }
26   </script>
```

8.2 TodoList 앱에 <script setup> 적용하기

❖ 예제 09-16 : src/components/TodoList.vue 변경

```
1   <template>
2     <div class="row">
3       <div class="col">
4         <ul class="list-group">
5           <TodoListItem v-for="todoItem in todoList" :key="todoItem.id"
6             :todoItem="todoItem" @delete-todo="emit('delete-todo', todoItem.id)"
7             @toggle-completed="emit('toggle-completed', todoItem.id)" />
8         </ul>
9       </div>
10    </div>
11  </template>
12
13  <script setup>
14    import TodoListItem from './TodoListItem.vue'
15
16    const props = defineProps({
17      todoList : { type : Array, required:true }
18    })
19    const emit = defineEmits(['delete-todo','toggle-completed'])
20  </script>
```

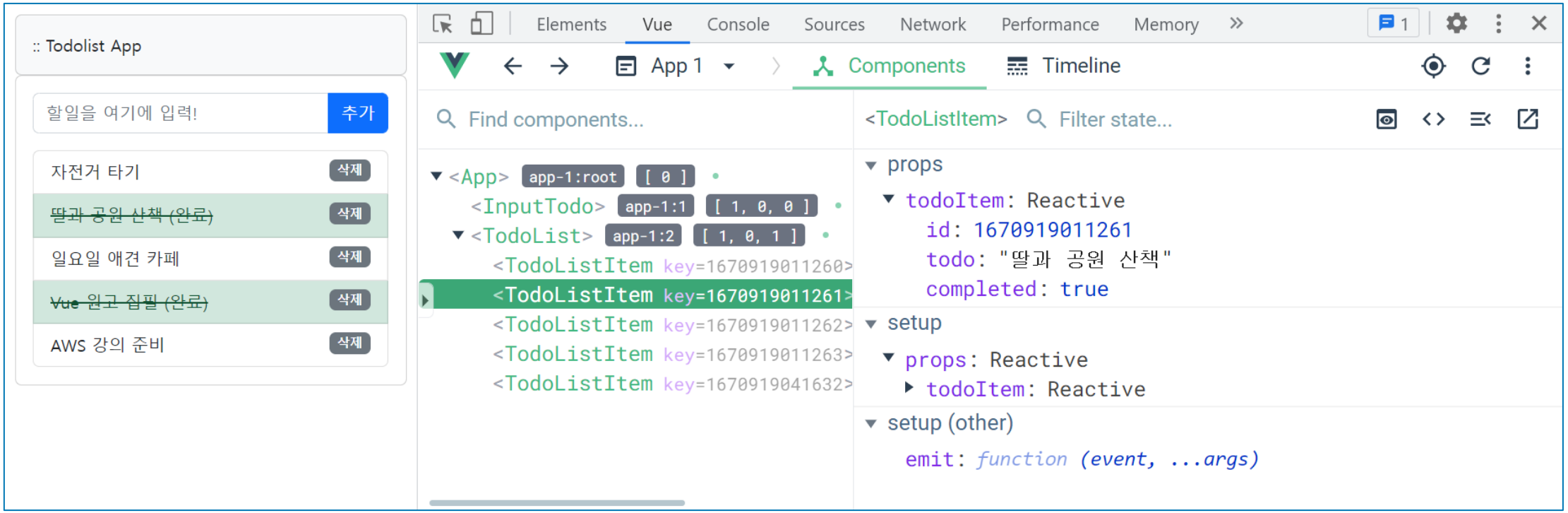
8.2 TodoList 앱에 <script setup> 적용하기

❖ 예제 09-17 : src/components/TodoListItem.vue 변경

```
1   <template>
2     <li class="list-group-item"
3       :class="{ 'list-group-item-success': todoItem.completed } "
4       @click="emit('toggle-completed', todoItem.id)" >
5       <span class="pointer" :class="{ 'todo-done': todoItem.completed }">
6         {{todoItem.todo}} {{ todoItem.completed ? "(완료)" : "" }}
7       </span>
8       <span class="float-end badge bg-secondary pointer"
9         @click.stop="emit('delete-todo', todoItem.id)">삭제</span>
10    </li>
11  </template>
12
13  <script setup>
14    const props = defineProps({
15      todoItem : { type : Object, required: true }
16    })
17    const emit = defineEmits(['delete-todo','toggle-completed'])
18  </script>
```

8.2 TodoList 앱에 <script setup> 적용하기

❖ 실행 결과



9. 마무리

지금까지 Vue 3에 추가된 Composition API에 대해 살펴보았습니다. Composition API를 반드시 사용할 필요는 없지만 기존 옵션 API보다 장점이 많으므로 권장합니다. 특히 새롭게 Vue 애플리케이션을 개발한다면 Composition API 사용을 적극 권장합니다.

특히 Composition API를 적용할 때 반응성 데이터(reactive, ref)를 직접 만들어야 하고, 반응성을 잃지 않도록 하기 위해 주의할 부분들이 있으니 9.3의 반응성 데이터 부분을 참조하여 확인하세요.