Flux Architecture

with JavaFX

Manuel Mauky
@manuel_mauky

Saxonia Systems
So geht Software.

Manuel Mauky

Software Developer at Saxonia Systems AG

Living in Görlitz, Germany

JUG-Leader of JUG Görlitz

- Frontend Development (JSF, JavaFX, JavaScript)
- Functional Programming

What is Flux?

“Flux is the application architecture that Facebook uses for building client-side web applications.”

https://facebook.github.io/flux/

What is Flux?

- Frontend architecture pattern

What is Flux?

- Frontend architecture pattern
- Core idea: **unidirectional data flow**

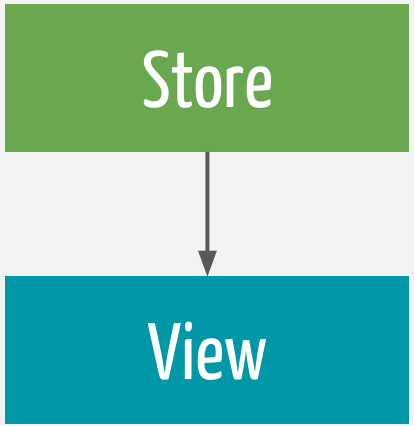
What is Flux?

- Frontend architecture pattern
- Core idea: **unidirectional data flow**
- Goal: easier understanding of what is happening in the application

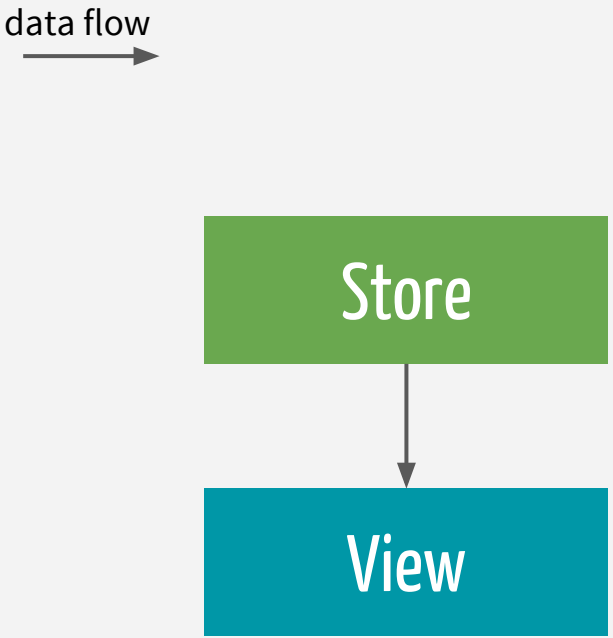
Store

- application state
- logic
- represents a business unit

data flow



- View visualizes data from the Store
- When the data in the Store changes the View will update itself accordingly

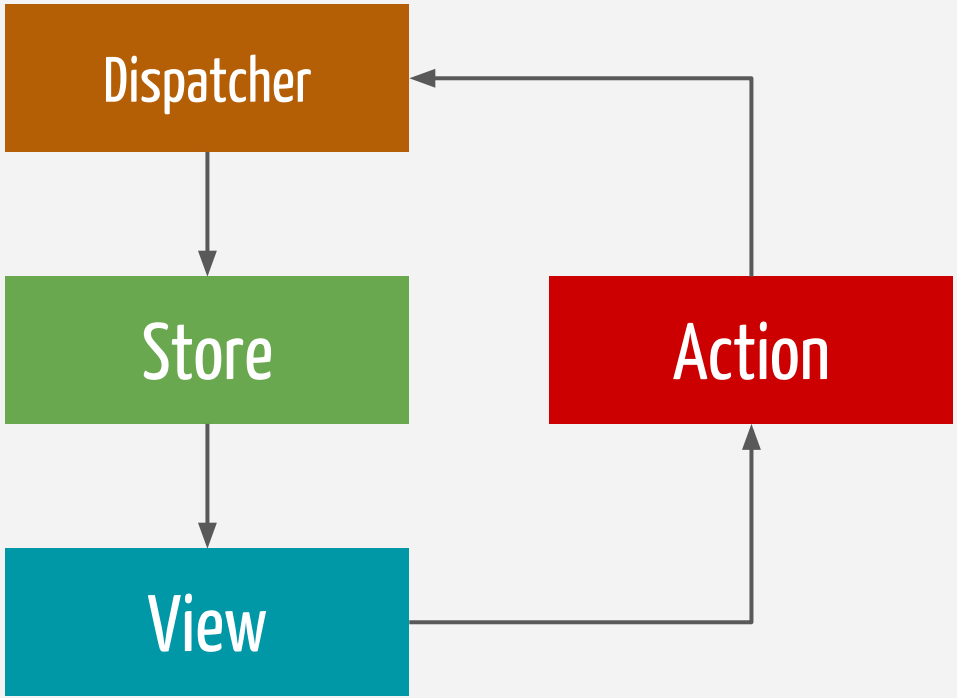

data flow
Store
View

data flow

- View generates “Actions” based on user interaction
- Action has a type and (optional) payload data
- similar to Command Pattern

Example: Actions

```
{
    type: "CREATE_USER_ACTION",
    payload: {
        username: "Luise",
        email: "luise@example.org"
    }
}
```

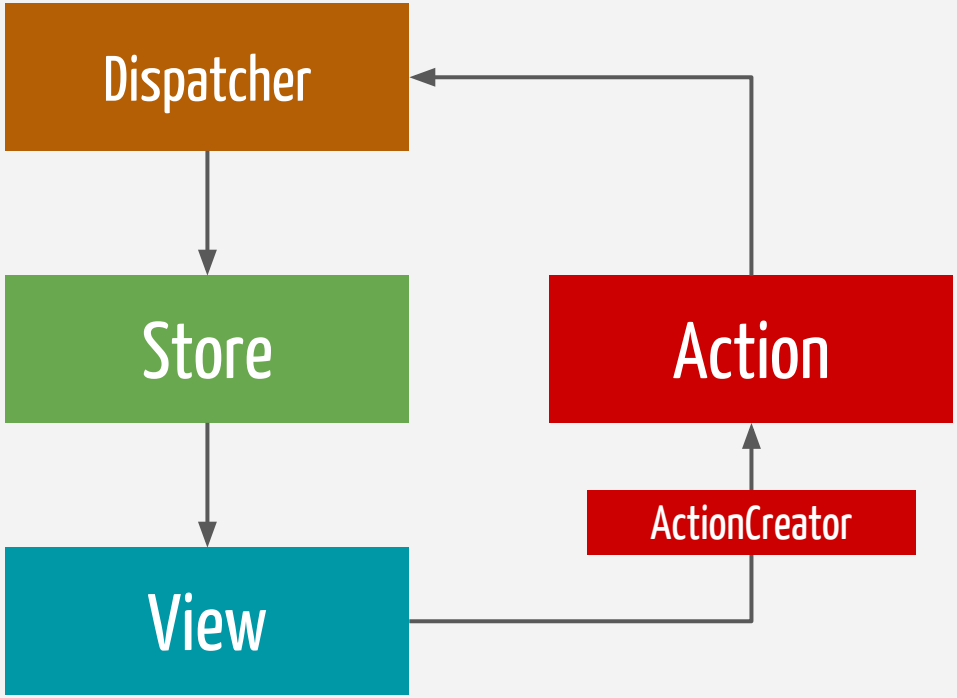


- Dispatcher passes **all** Actions to **all** Stores
- Each Store decides if and how it will react to actions

Flux Architecture

- Store: no "setters" - can't be manipulated from the outside
- explicit modelling of business actions
- Dispatcher works synchronously: Actions are handled one at a time
- Optional: dependencies between Stores → order of action handling can be controlled

Flux: Details

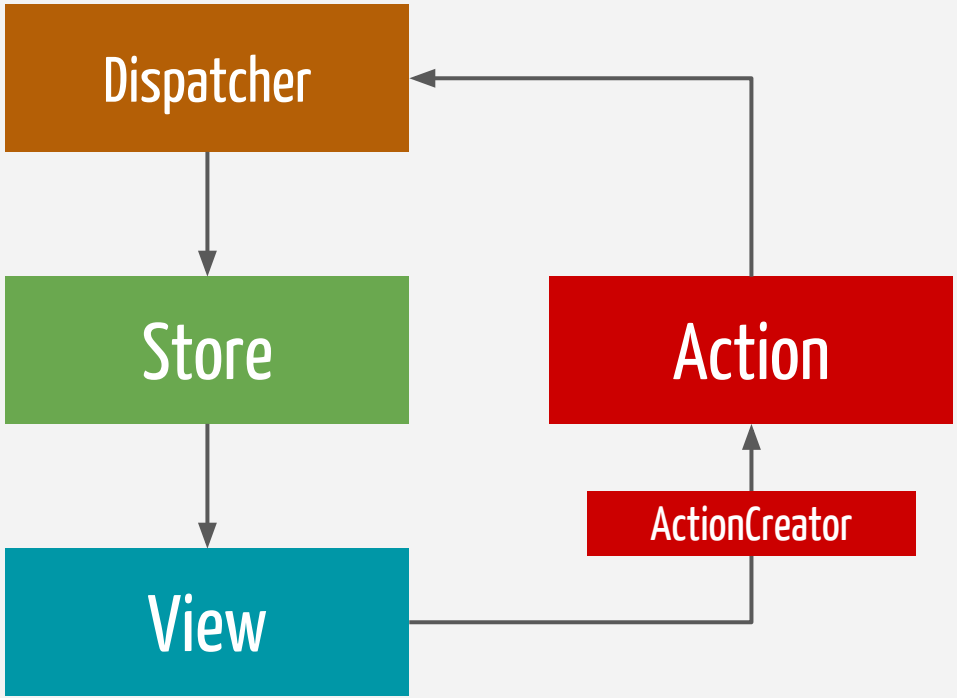


- ActionCreator: reusable function that creates actions
- used by View

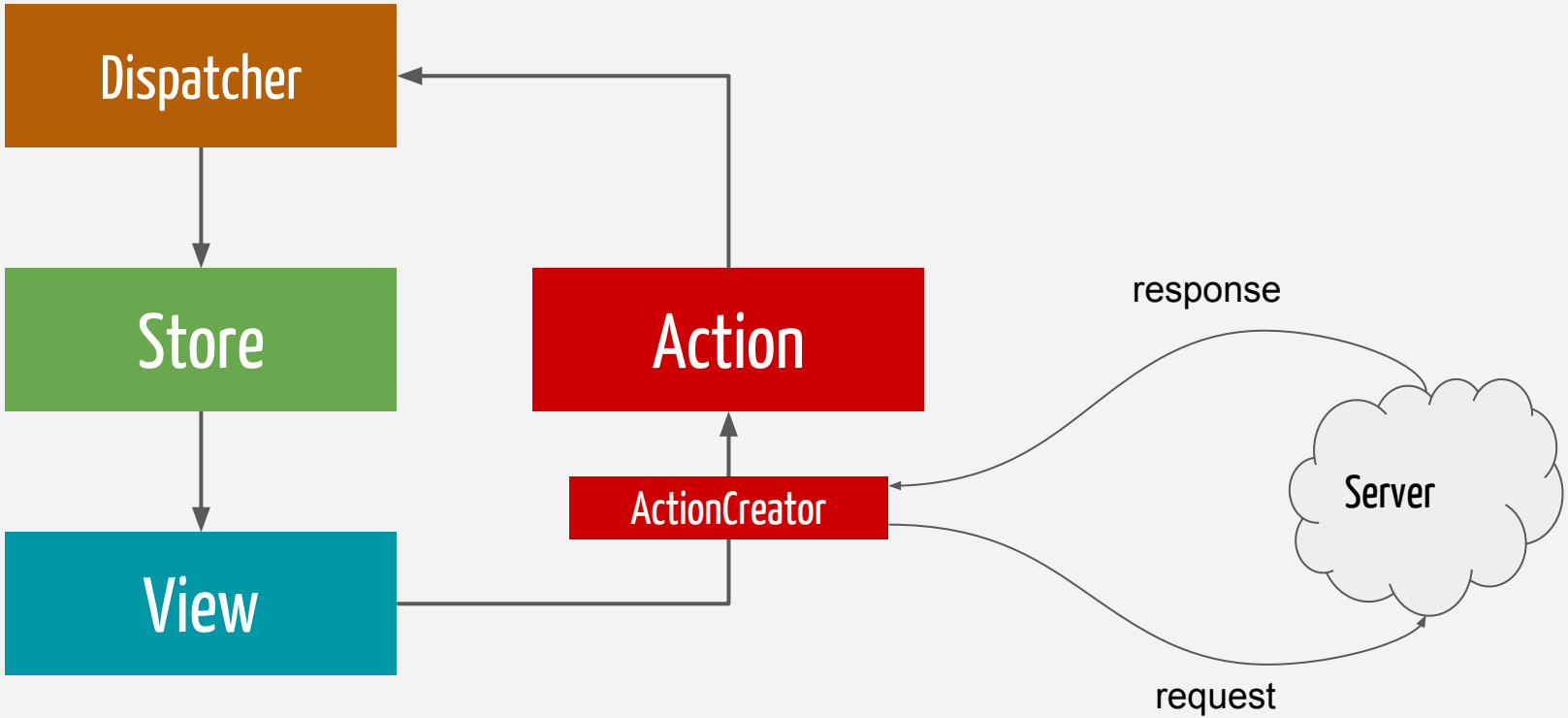
Example: ActionCreator

```
const createUser = (username, email) => {
    dispatch({
        type: "CREATE_USER_ACTION",
        payload: {
            username: username,
            email: email
        }
    });
};
```

How to handle async interactions? Like REST requests?



How to handle async interactions? Like REST requests?

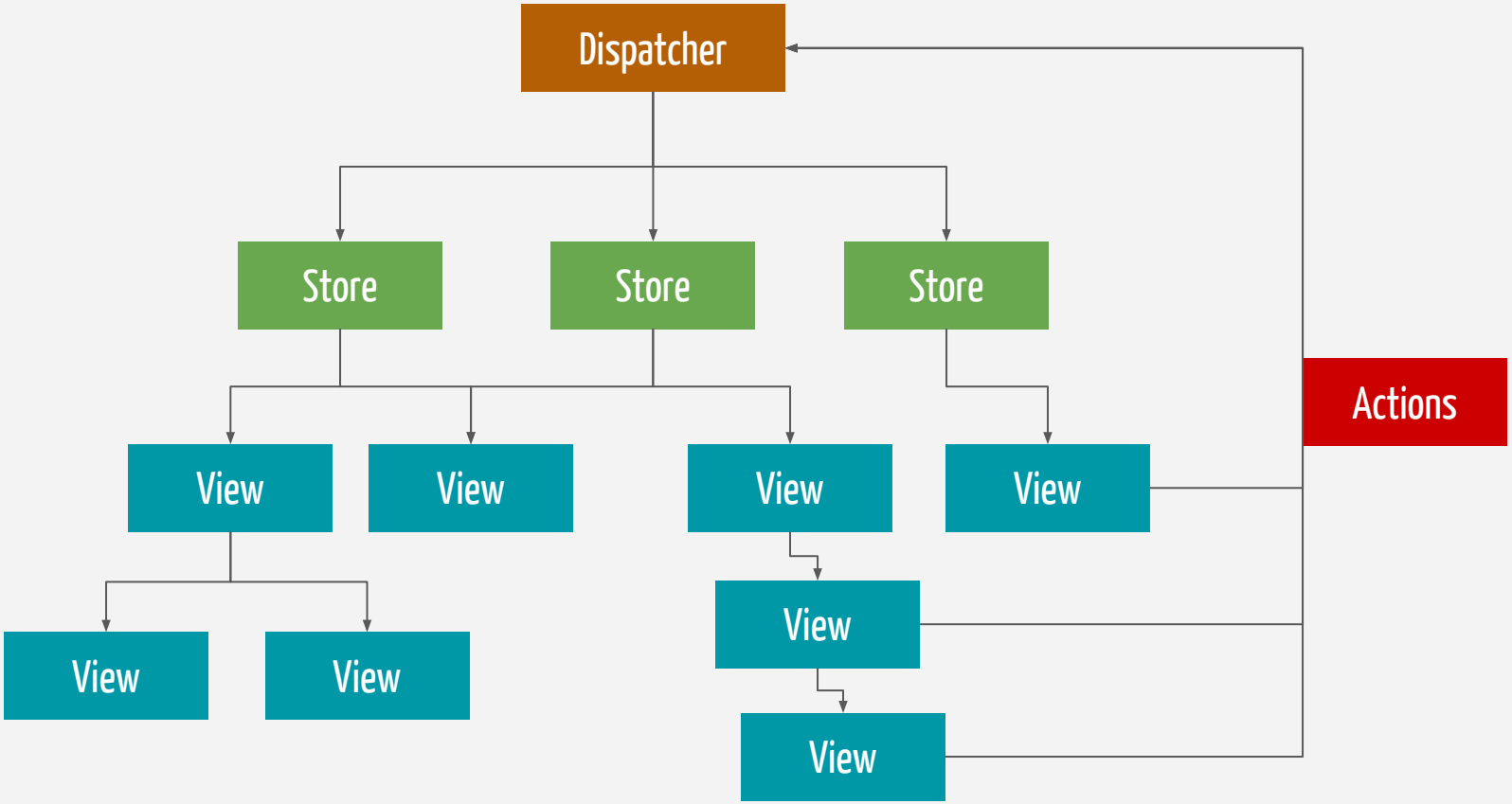


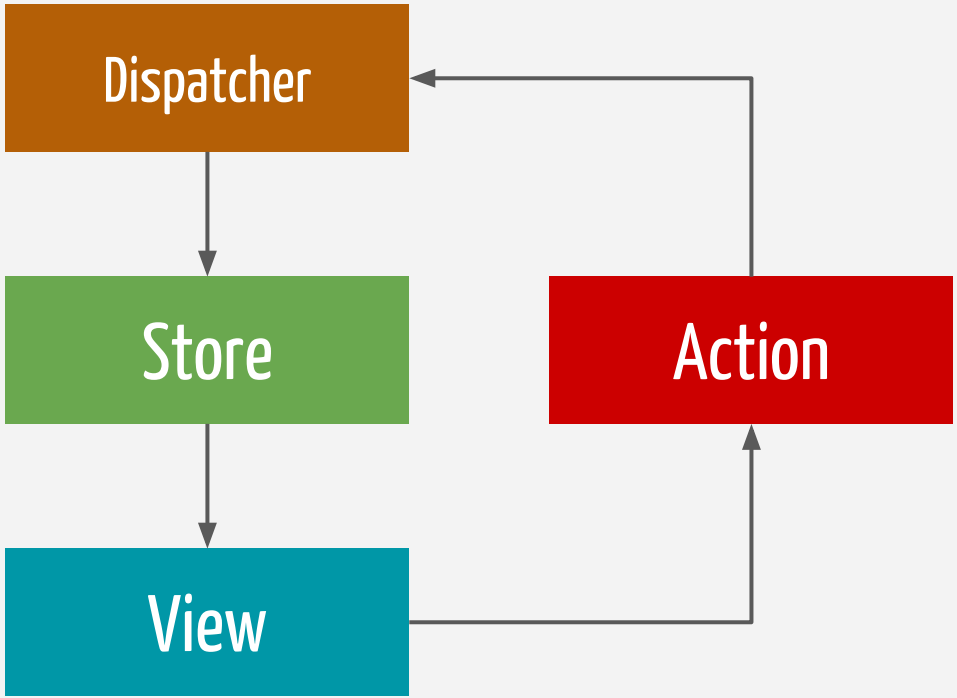
How to handle async interactions? Like REST requests?

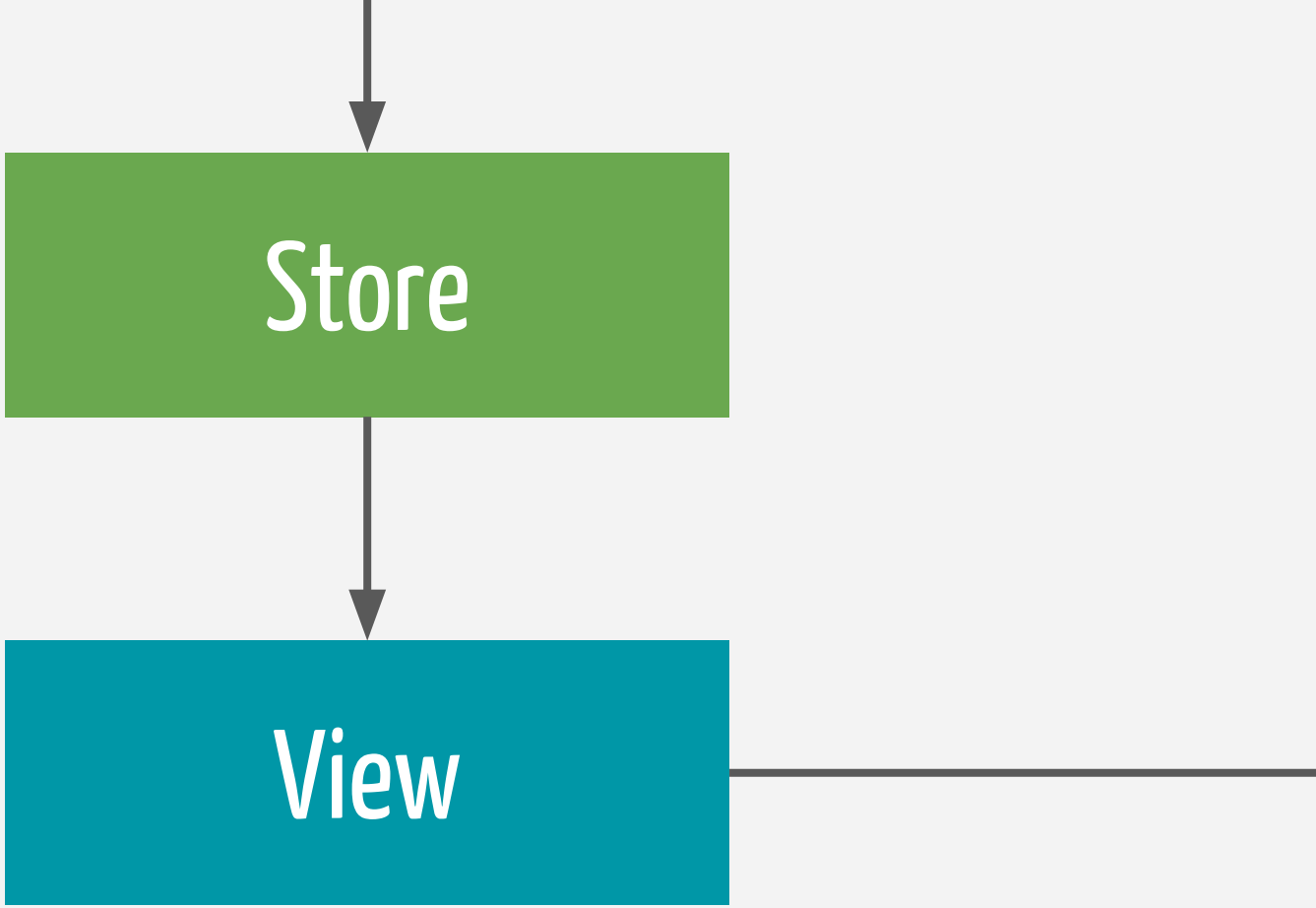
ActionCreator can be asynchronous

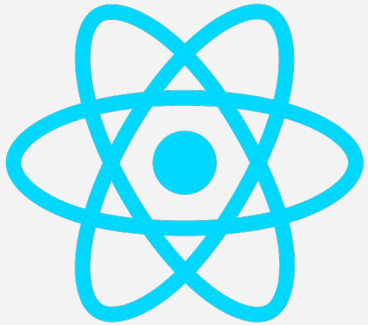
1. Dispatch Action "FETCH_DATA_STARTED"
2. Get data from Server
3. When data arrives → dispatch Action: "FETCH_DATA_SUCCESSFUL"
4. If timeout or error happens → dispatch Action that represents error: "FETCH_DATA_FAIL_TIMEOUT"

```
const fetchUsers = () => {
    dispatch({type: "FETCH_USERS_STARTED"});

    fetch("http://my.api.example.com/users")
        .then(response => response.json)
        .then(json => dispatch({
                type: "FETCH_USERS_SUCCESSFUL",
                payload: json
            }),
            error => dispatch({
                type: "FETCH_USERS_FAILED"
            })
        );
};
```


Dispatcher
Store
Store
Store
Actions
View
View
View
View
View
View
View
View


Dispatcher
Store
View
Action


Store
View



Short introduction: **React.JS**

```
var React = require("react");

var HelloWorld = React.createClass({
    render: function() {
        return (
            <div>
                <p>Hello World!</p>
            </div>
        );
    }
});
```

```
import React from "react";

class HelloWorld extends React.Component {
    render() {
        return (
            <div>
                <p>Hello World!</p>
            </div>
        )
    }
}
```

```
import React from "react";

const HelloWorld = () => (
    <div>
        <p>Hello World!</p>
    </div>
);
```

```
import React from "react";

const HelloWorld = () => (
    <div>
        <p>Hello World!</p>
    </div>
);


// Usage
<HelloWorld />
```

```
import React from "react";

const HelloWorld = (props) => (
    <div>
        <p>Hello {props.subject}</p>
    </div>
);


// Usage
<HelloWorld subject="World"/>
```

Composition

```
class Hellos extends React.Component {
    render () {
        <ul>
            <li><HelloWorld subject="World" />
            <li><HelloWorld subject="Moon" />
            <li><HelloWorld subject="Sun" />
        </ul>
    }
};
```

```
class Greeter extends React.Component {
    constructor() {
        this.state = { counter: 0 }
    }
    count() {
        const oldValue = this.state.counter;
        this.setState({ counter: oldValue + 1 });
    }
    render() {
        return (
            <div>
                <p>Value: {this.state.counter}</p>
                <button onClick={this.count}>+</button>
            </div>
        )
    }
}
```

React.JS

- every state change leads to rerendering of the whole component (incl. subcomponents)
- kind of functional programming
- easy programming model

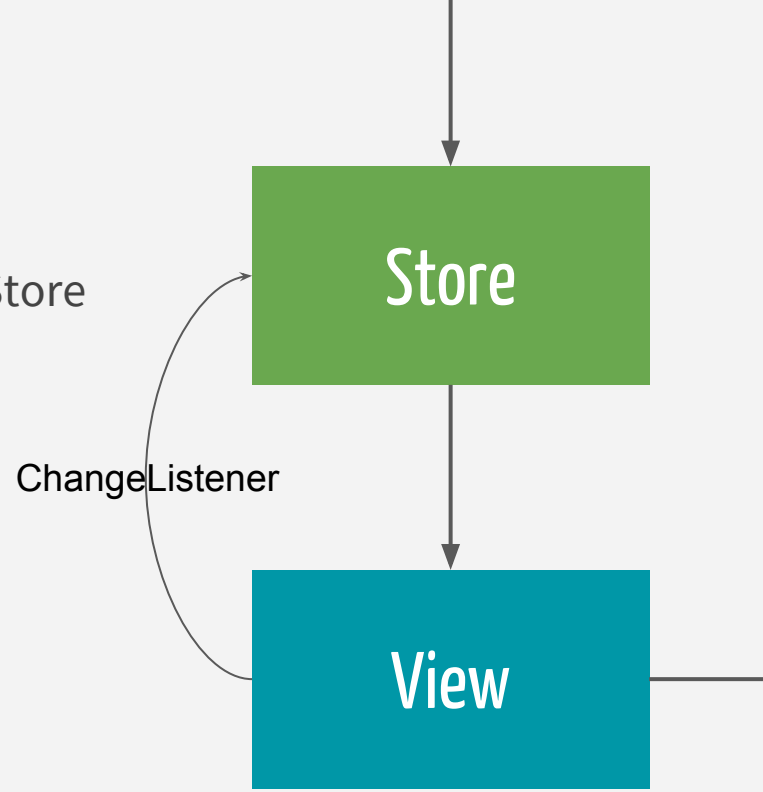
React.JS

- every state change leads to rerendering of the whole component (incl. subcomponents)
- kind of functional programming
- easy programming model

- Virtual-DOM
- not the "real" DOM is rerendered but only the virtual DOM
- efficient diff algorithm calculates minimal necessary "real" DOM operations

Flux with React.JS

- parent component registers ChangeListener on Store
- in the ChangeListener:
  - get data from Store
  - pass data to setState method
  - component gets rerendered



Flux with JavaFX?

Naive Implementation: Do it the same way

Problems:

- JavaFX has no Virtual-DOM/Virtual-SceneGraph (at least there is no such API for developers)
- complete rerendering on every change is slow and inefficient
- not fully declarative

JavaFX is not fully declarative

React:

```
const list = () => (
    <ul>
        <li>Hello</li>
        <li>World</li>
    </ul>
)
```

JavaFX:

```
<VBox fx:controller="Controller">
    <ListView fx:id="myList" />
</VBox>

class Controller {
    @FXML
    private ListView myList;
    public void initialize() {
        myList.getItems().add("Hello");
        myList.getItems().add("World");
    }
}
```

Naive Implementation: Do it the same way

Problems:

- JavaFX has no Virtual-DOM/Virtual-SceneGraph (at least there is no such API for developers)
- complete rerendering on every change is slow and inefficient
- not fully declarative

However:

- JavaFX is ready for *Reactive Programming*
  - data binding
  - reactive streams (*RxJavaFX* or *ReactFX*) // "React**FX**" has nothing to do with "React.**JS**"

Idea

- Stores use properties internally
- Stores provide read-only properties to the outside world
- View binds to read-only properties of the Store
- Action as classes (Command Pattern)
- Dispatcher as reactive stream
- Stores subscribe to Dispatcher stream

Example: Counter

```
public class CounterStore {

    private IntegerProperty counter = new SimpleIntegerProperty();


    public ReadOnlyIntegerProperty counterValue() {
        return counter;
    }
}
```

```
public class CounterView {

    @FXML
    private Label valueLabel;

    @Inject
    private CounterStore store;

    public void initialize() {
        valueLabel.textProperty().bind(store.counterValue());
    }
}
```

```
public class IncreaseAction implements Action {

    private final int amount;

    public IncreaseAction(int amount) {
        this.amount = amount;
    }

    public int getAmount() {
        return amount;
    }
    // equals & hashcode
}
```

```
import org.reactfx.EventSource;
import org.reactfx.EventStream;

public class Dispatcher {

    private EventSource<Action> actionStream = new EventSource<>();


    public void dispatch(Action action) {
        actionStream.push(action);
    }

    public EventStream<Action> getActionStream() {
        return actionStream;
    }
}
```

```
public class CounterView implements View {

    ...

    @FXML
    public void increaseButtonPressed () {
        Dispatcher.getInstance().dispatch(new IncreaseAction(1));
    }

}
```

```
public class CounterStore {

    private IntegerProperty counter = new SimpleIntegerProperty();

    public CounterStore() {
        Dispatcher.getInstance()
            .getActionStream()
            .filter(a -> a.getClass().equals(IncreaseAction.class))
            .map(a -> (IncreaseAction) a)  // cast is needed
            .subscribe(this::increase);
    }

    private void increase(IncreaseAction action) {
        counter.set(counter.get() + action.getAmount());
    }

    public ReadOnlyIntegerProperty counterValue() {
        return counter;
    }
}
```

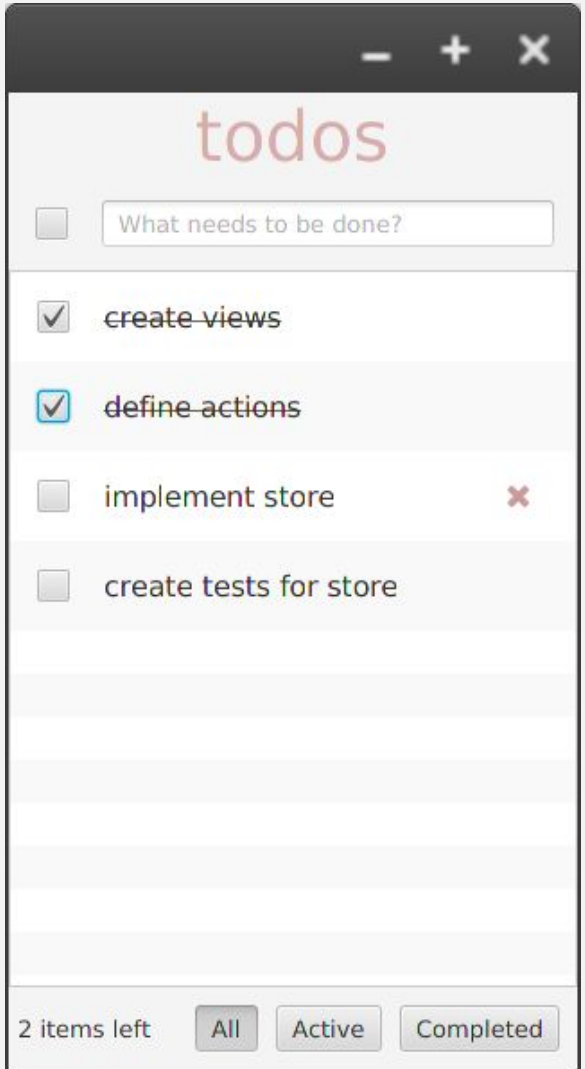
```
public abstract class Store {

    protected <T extends Action> void subscribe(
            Class<T> type, Consumer<T> consumer) {

        Dispatcher.getInstance()
            .getActionStream()
            .filter(a -> a.getClass().equals(type))
            .map(a -> (T) a)
            .subscribe(consumer);
    }
}
```

```
public class CounterStore extends Store {

    private IntegerProperty counter = new SimpleIntegerProperty();

    public CounterStore() {
        subscribe(IncreaseAction.class, this::increase);
    }

    private void increase(IncreaseAction action) {
        counter.set(counter.get() + action.getAmount());
    }

    public ReadOnlyIntegerProperty counterValue() {
        return counter;
    }
}
```

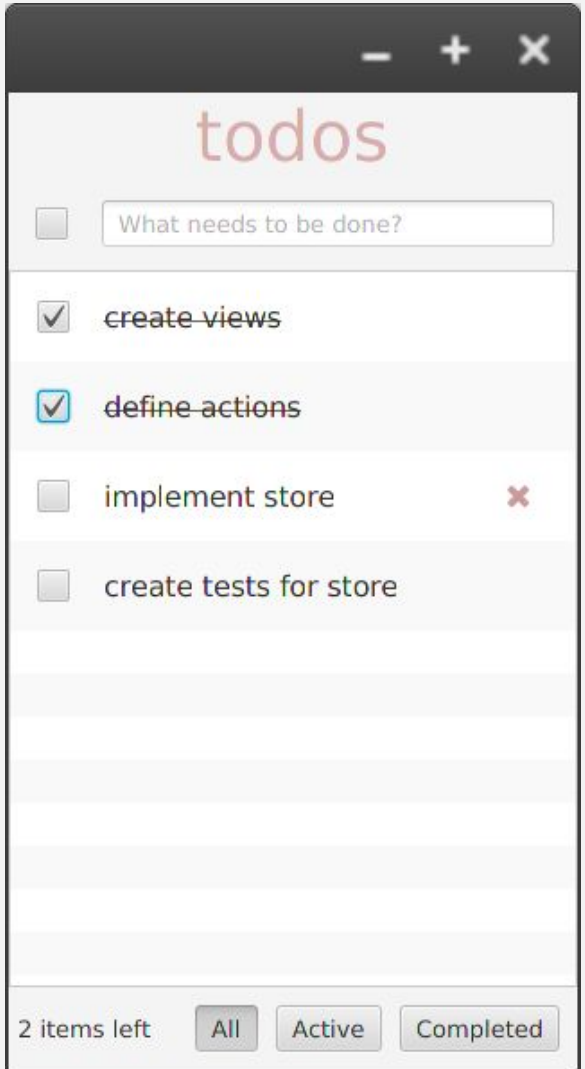
```
@Test
public void test() {
    // given
    assertThat(store.counter()).hasValue(0);

    // when
    Dispatcher.getInstance().dispatch(new IncreaseAction(1));

    // then
    assertThat(store.counter()).hasValue(1);
}
```

Details, Benefits & Difficulties

Example: Todo List app



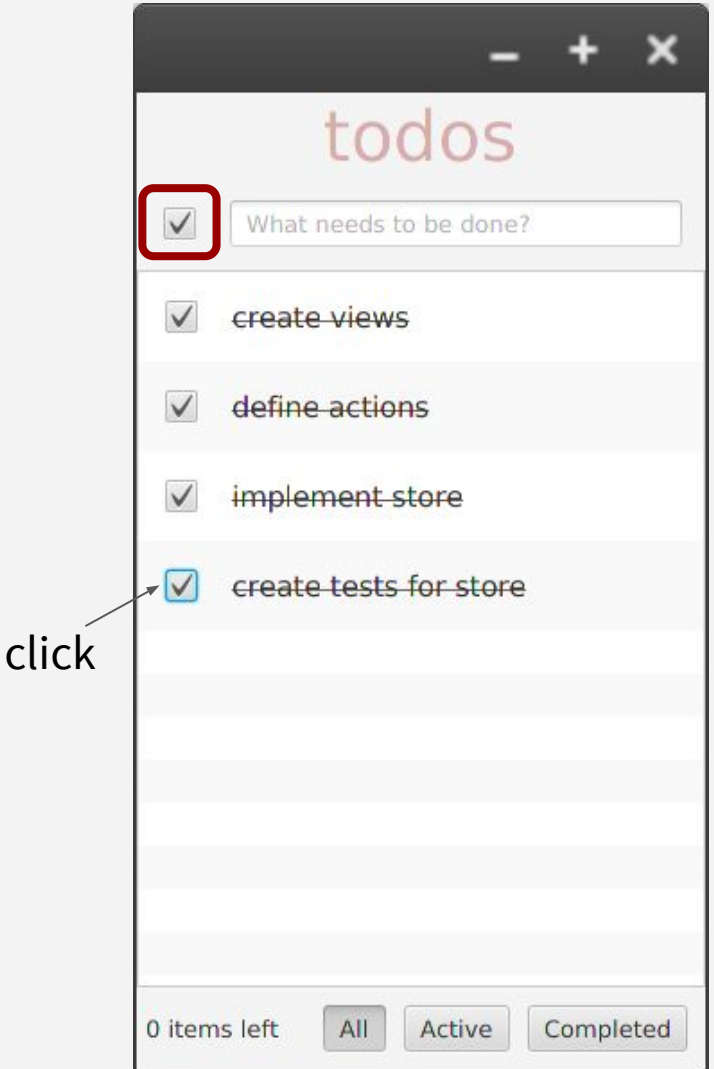
CheckBox behavior



CheckBox behavior

when all items are checked,

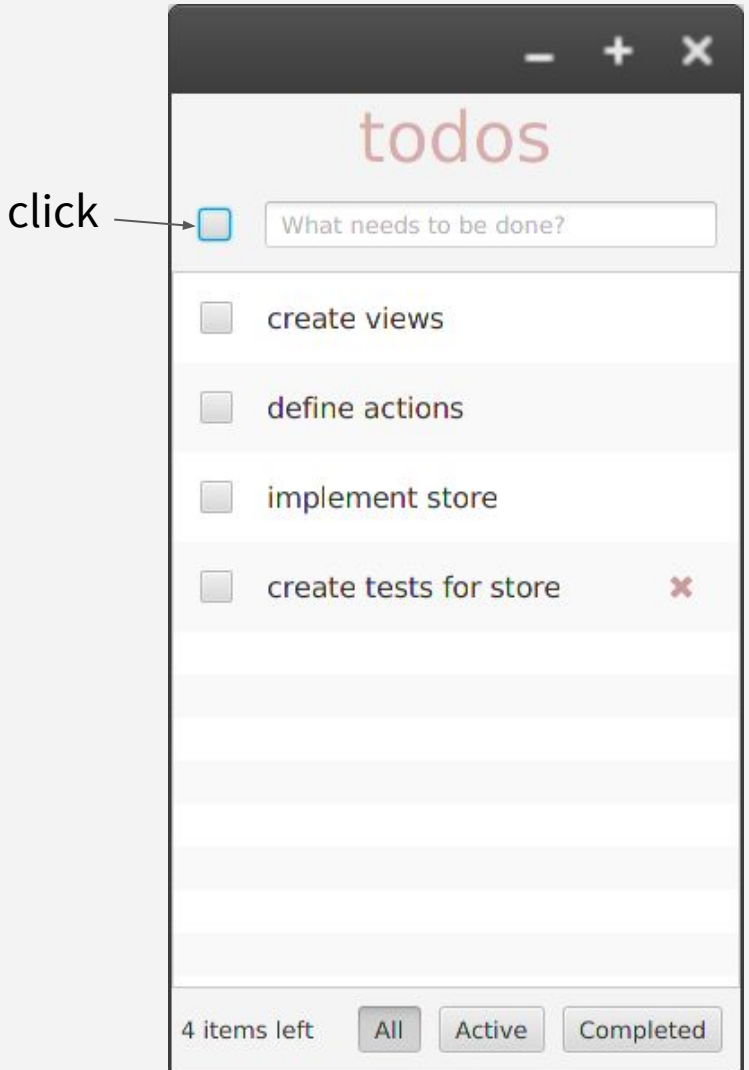
the top checkbox should switch too



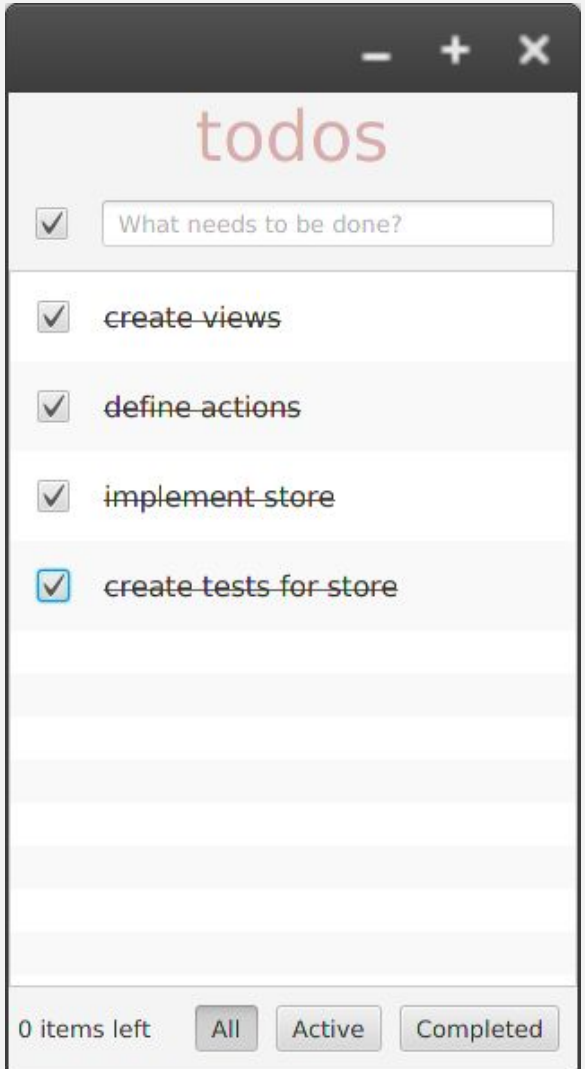
CheckBox behavior

When the top checkbox is clicked,

all items should be checked/unchecked



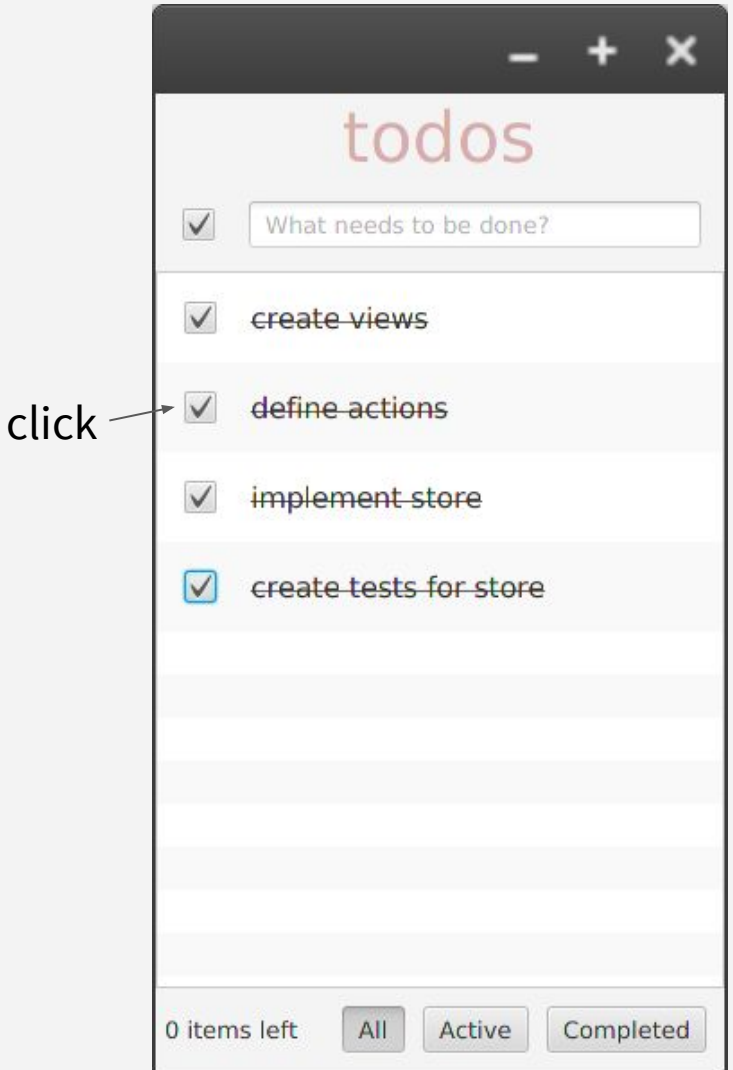
CheckBox behavior



CheckBox behavior

When all items are checked and

a single checkbox is clicked...

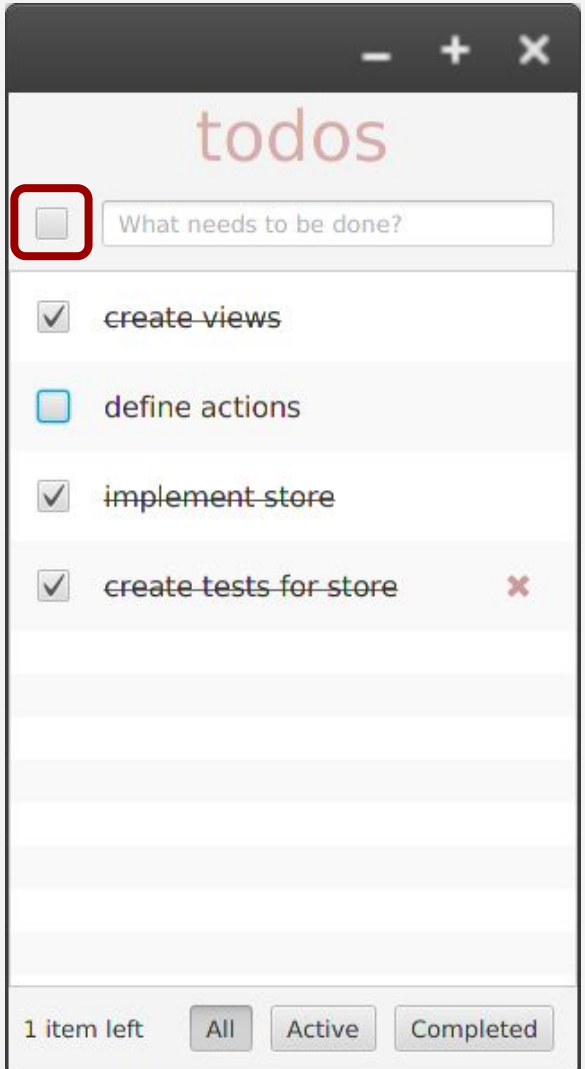


CheckBox behavior

When all items are checked and

a single checkbox is clicked,

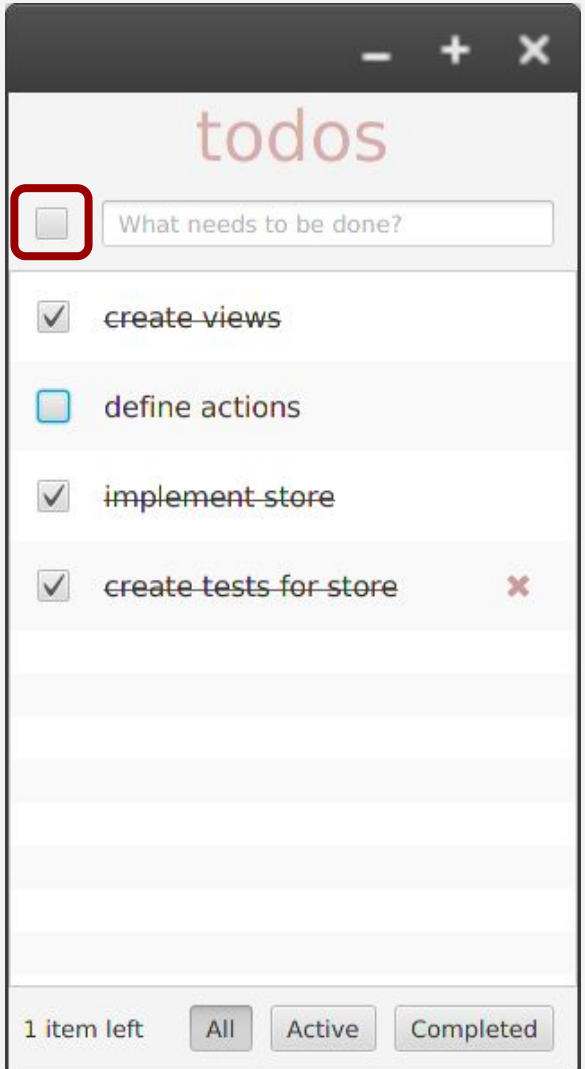
the top checkbox should switch to unchecked



CheckBox behavior

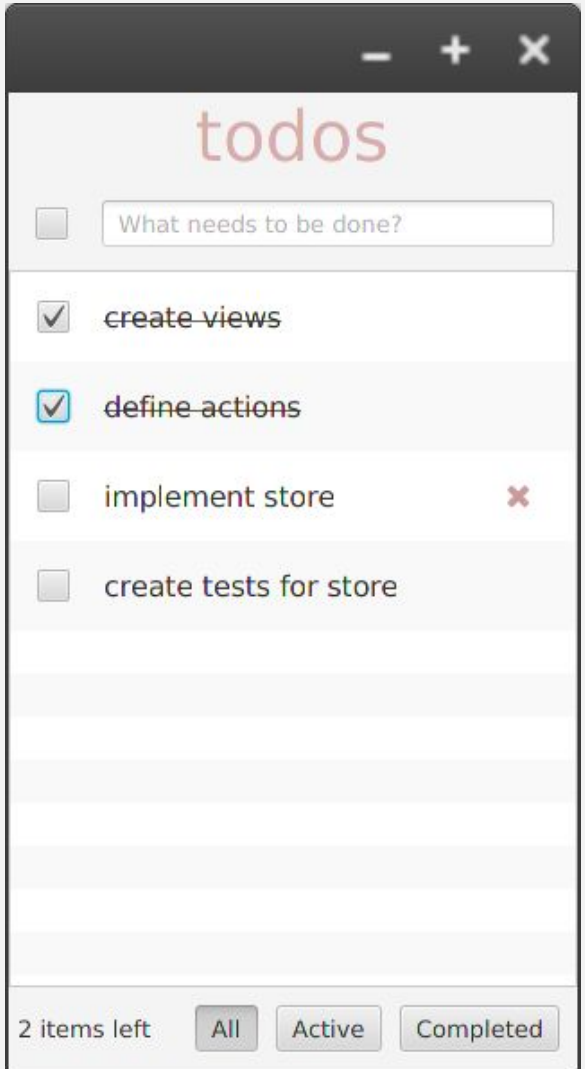
Top checkbox combines two tasks:

- shows status of items
- manipulate status of all items



CheckBox behavior

How to build this only by looking at the state?

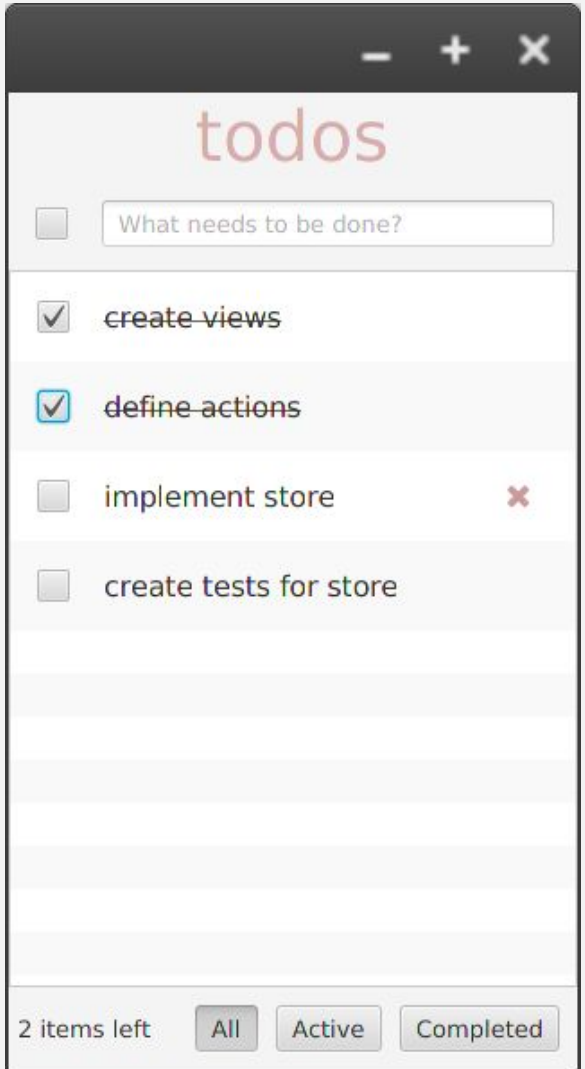


CheckBox behavior

How to build this only by looking at the state?

Flux: 2 distinct action types:

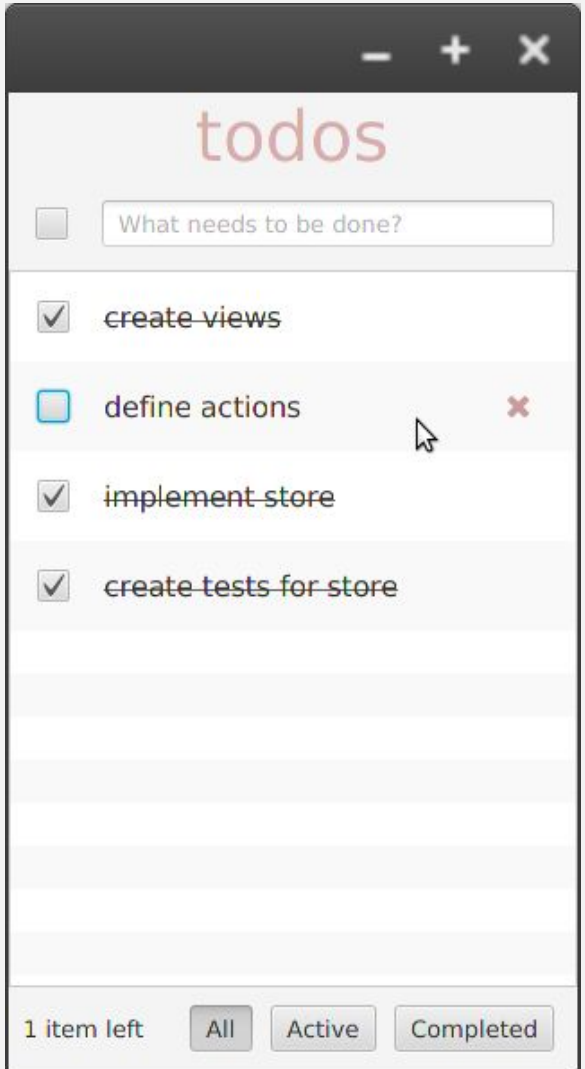
- ClickTopCheckboxAction
- ClickItemCheckboxAction



Stateful Views?

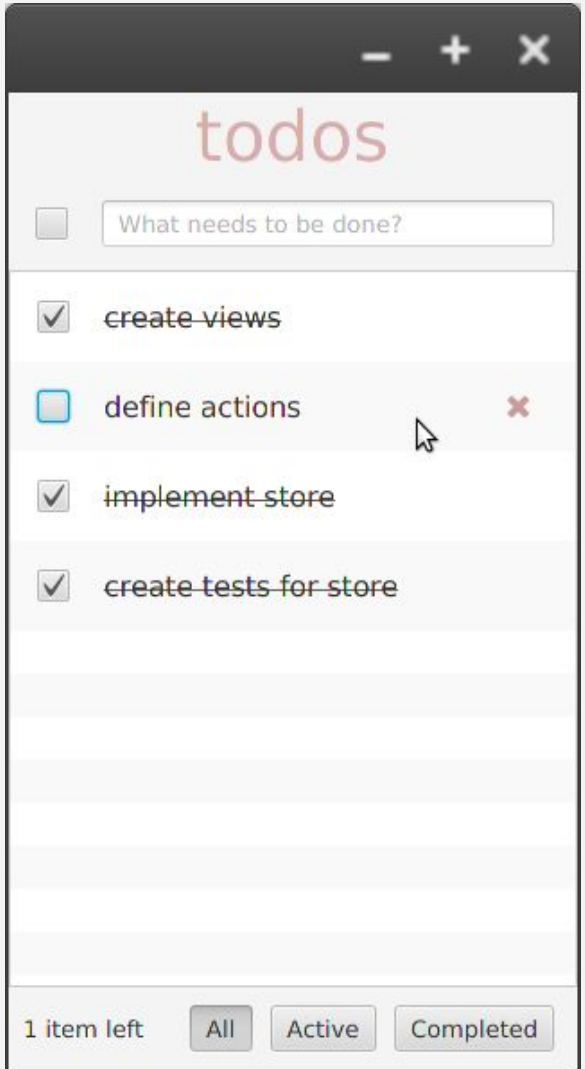
Stateful Views?

- delete button is only visible on mouse over



Stateful Views?

- delete button is only visible on mouse over
- OnMouseOverAction?
- OnMouseOverFinishedAction?

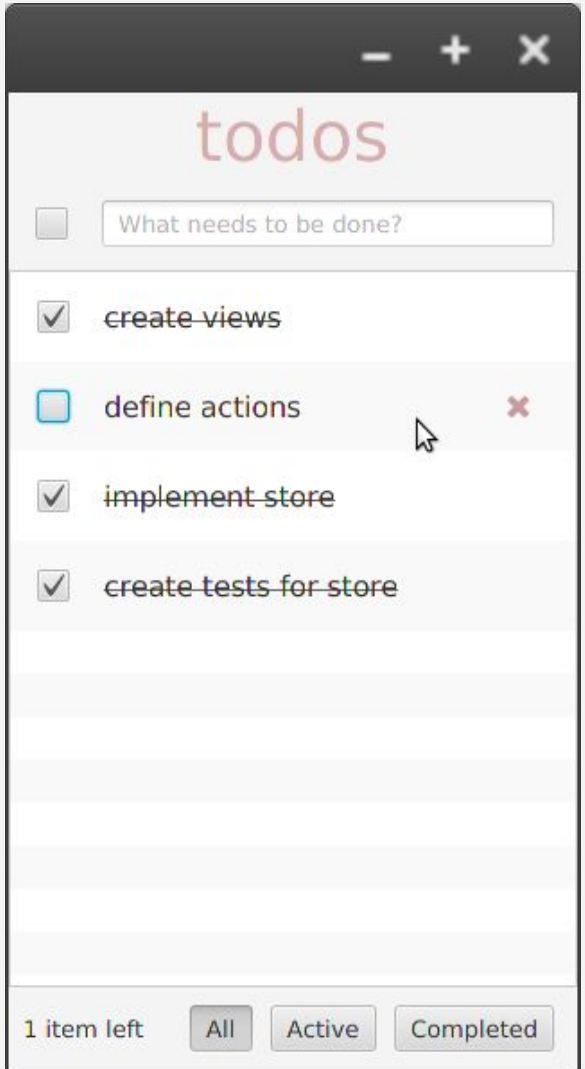


Stateful Views?

- delete button is only visible on mouse over
- OnMouseOverAction?
- OnMouseOverFinishedAction?

or simply:

```
deleteButton.visibleProperty()

        .bind(root.hoverProperty);
```



Benefits

Benefits

- No cascading updates → easier reasoning what's happening

Benefits

- No cascading updates → easier reasoning what's happening
- Find out where a bug is located:
  - Given a state in the Store: does the UI render correctly?
  - Given a state: when an action is incoming, does the state change as expected?
  - Given an user interaction: Is the correct action dispatched?

Benefits

- No cascading updates → easier reasoning what's happening
- Find out where a bug is located:
  - Given a state in the Store: does the UI render correctly?
  - Given a state: when an action is incoming, does the state change as expected?
  - Given an user interaction: Is the correct action dispatched?
- Easy testing of frontend code

Benefits

- No cascading updates → easier reasoning what's happening
- Find out where a bug is located:
  - Given a state in the Store: does the UI render correctly?
  - Given a state: when an action is incoming, does the state change as expected?
  - Given an user interaction: Is the correct action dispatched?
- Easy testing of frontend code
- Undo/Redo is easier to implement

Benefits

- No cascading updates → easier reasoning what's happening
- Find out where a bug is located:
  - Given a state in the Store: does the UI render correctly?
  - Given a state: when an action is incoming, does the state change as expected?
  - Given an user interaction: Is the correct action dispatched?
- Easy testing of frontend code
- Undo/Redo is easier to implement
- Event Sourcing is easy to implement

Benefits

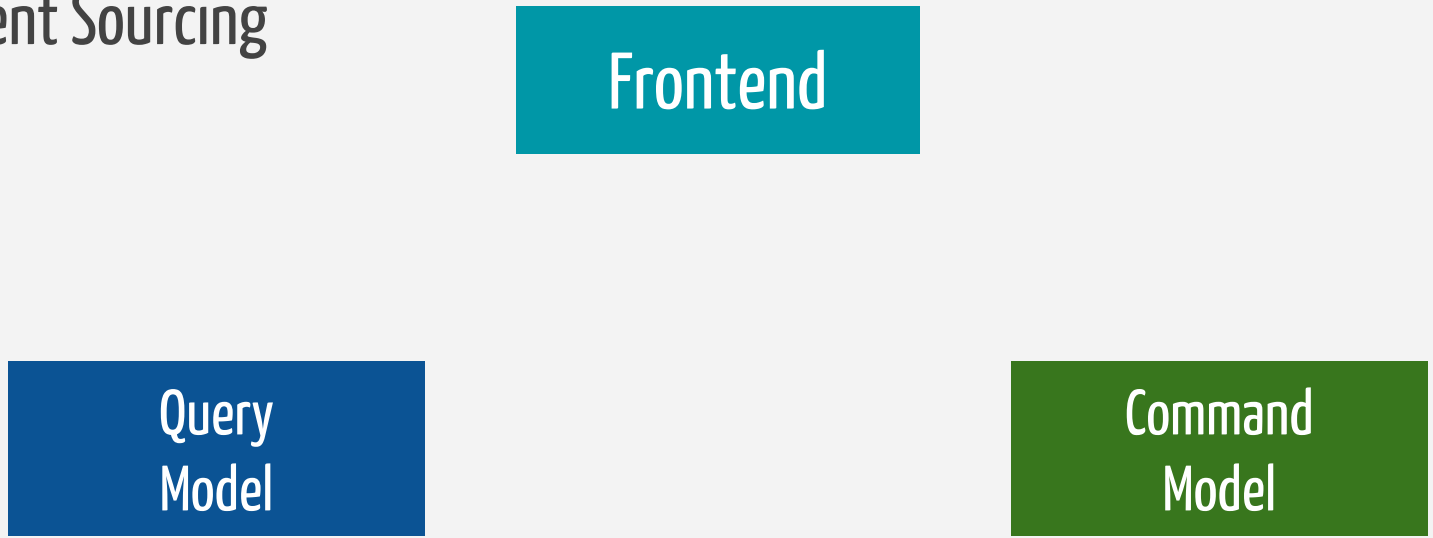
- No cascading updates → easier reasoning what's happening
- Find out where a bug is located:
  - Given a state in the Store: does the UI render correctly?
  - Given a state: when an action is incoming, does the state change as expected?
  - Given an user interaction: Is the correct action dispatched?
- Easy testing of frontend code
- Undo/Redo is easier to implement
- Event Sourcing is easy to implement
- more functional development style (not fully functional though)

Similar Ideas

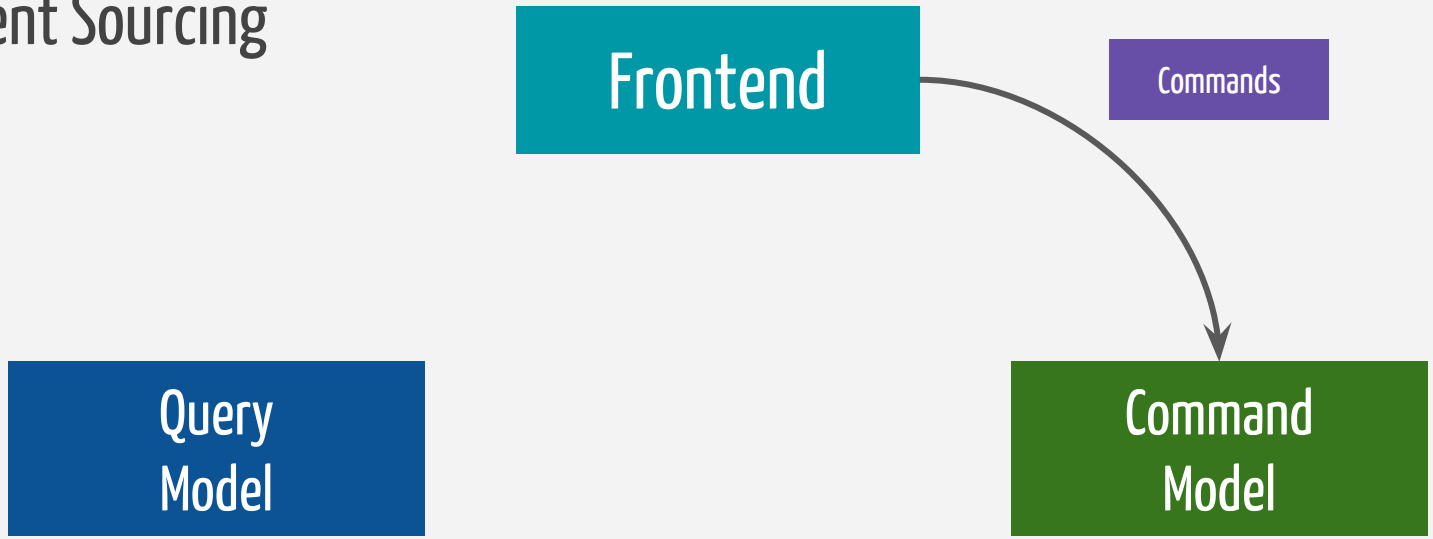
CQRS + Event Sourcing

Command Query Responsibility Segregation

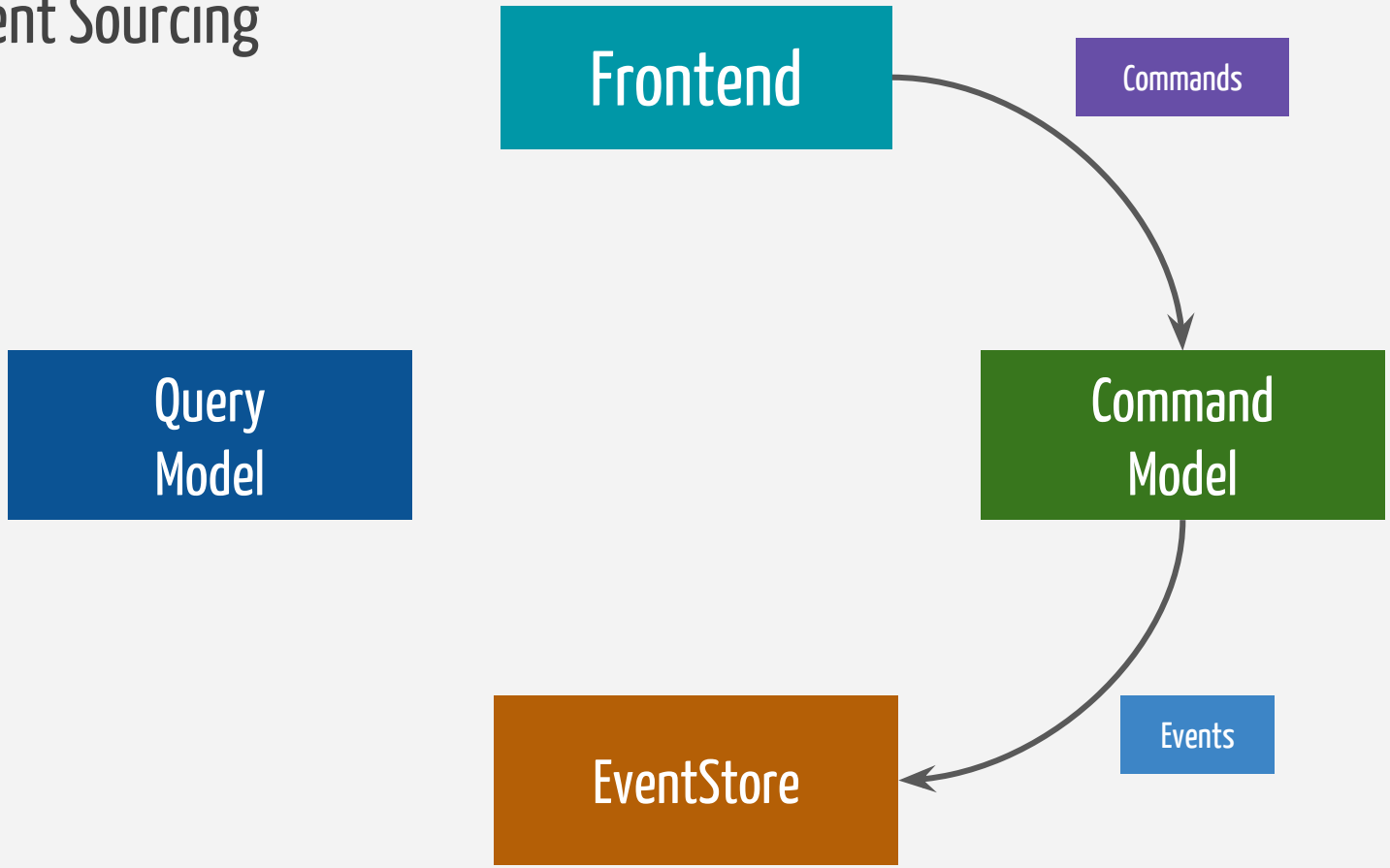
CQRS + Event Sourcing



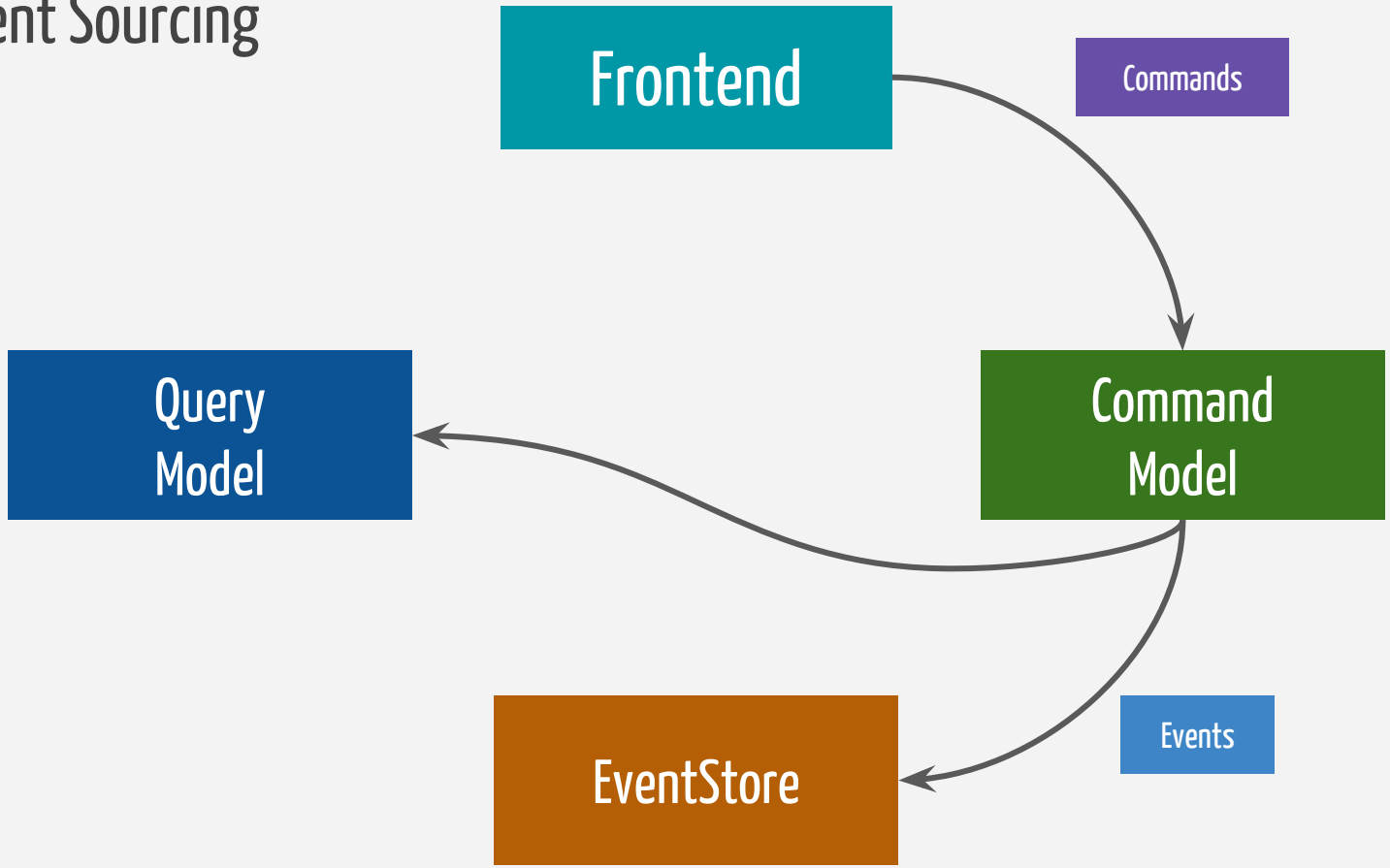
CQRS + Event Sourcing



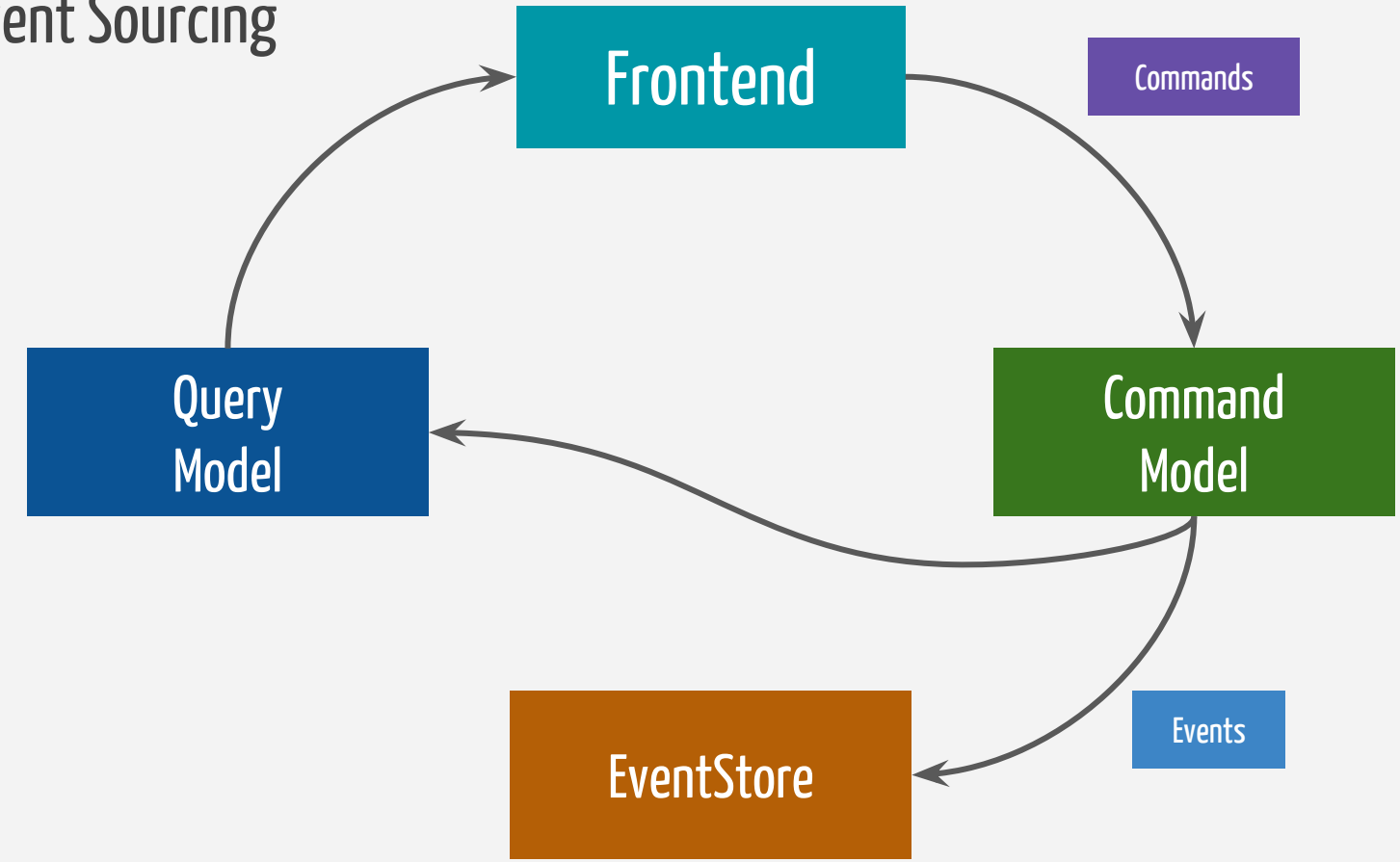
CQRS + Event Sourcing



CQRS + Event Sourcing



CQRS + Event Sourcing



CQRS + Event Sourcing

Similarities:

- command pattern
- unidirectional data flow

Differences:

- CQRS is an architecture pattern for whole application. Flux is "just" UI
- stricter separation between "Changing Data" and "Showing Data"
- other purposes: scaling different parts of application

Redux

Redux

- more functional version of Flux
- Single Store contains whole state
- Pure reducer function: `(oldState, action) -> newState`
- really cool with React.JS

Redux

- more functional version of Flux
- Single Store contains whole state
- Pure reducer function: `(oldState, action) -> newState`
- really cool with React.JS
- However:
  - Java isn't a functional language
  - immutable data structures missing :-(
  - declarative UI missing

Open Questions / Research to be done

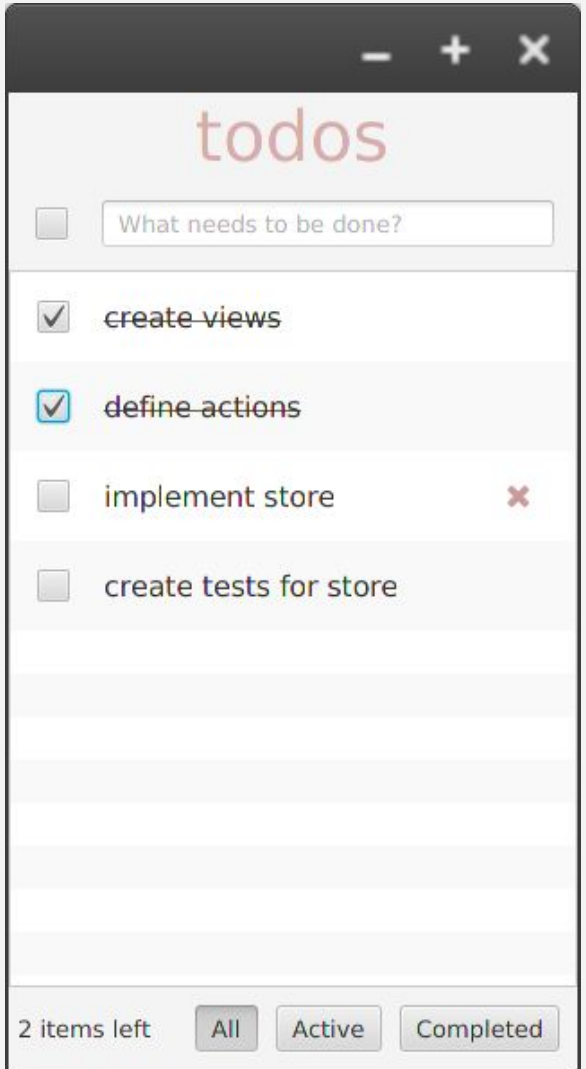
- Virtual-SceneGraph?
  - Promising: https://github.com/tferi/**templateFX**
  - declarative UI's for JavaFX inspired by React
  - Only Scala at the moment

Open Questions / Research to be done

- Virtual-SceneGraph?
  - Promising: https://github.com/tferi/**templateFX**
  - declarative UI's for JavaFX inspired by React
  - Only Scala at the moment
- Use functional languages? Groovy? Kotlin? **Frege**?

TodoMvcFX

- Like original TodoMVC but for JavaFX
- Same application with different frameworks
- Compare frameworks and patterns
- https://github.com/lestard/todomvcFX
- **Contributions are welcome!**



Q & A

Example Code:
https://github.com/lestard/FluxFX

Manuel Mauky

manuel.mauky@saxsys.de
http://lestard.eu
@manuel_mauky