React

FLUX

December 8, 2016

Agenda

1. FLUX base overview
2. FLUX Structure
3. Dispatcher
4. Actions/Action creators
5. Stores
6. Views

Flux base overview

Flux base overview

What is FLUX?

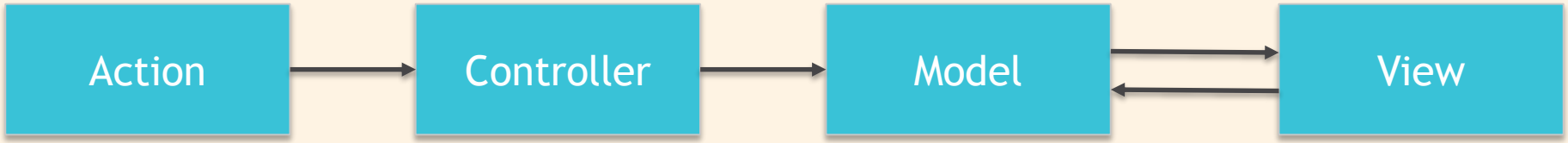
- **Flux** is the application architecture
- Created by Facebook and used for building client-side web applications.
- Complements React's composable view components by utilizing a **unidirectional data flow**.
- Solve problem of scale in MVC (MVVM) design pattern

Flux base overview

What’s wrong with MVC / MVVM?

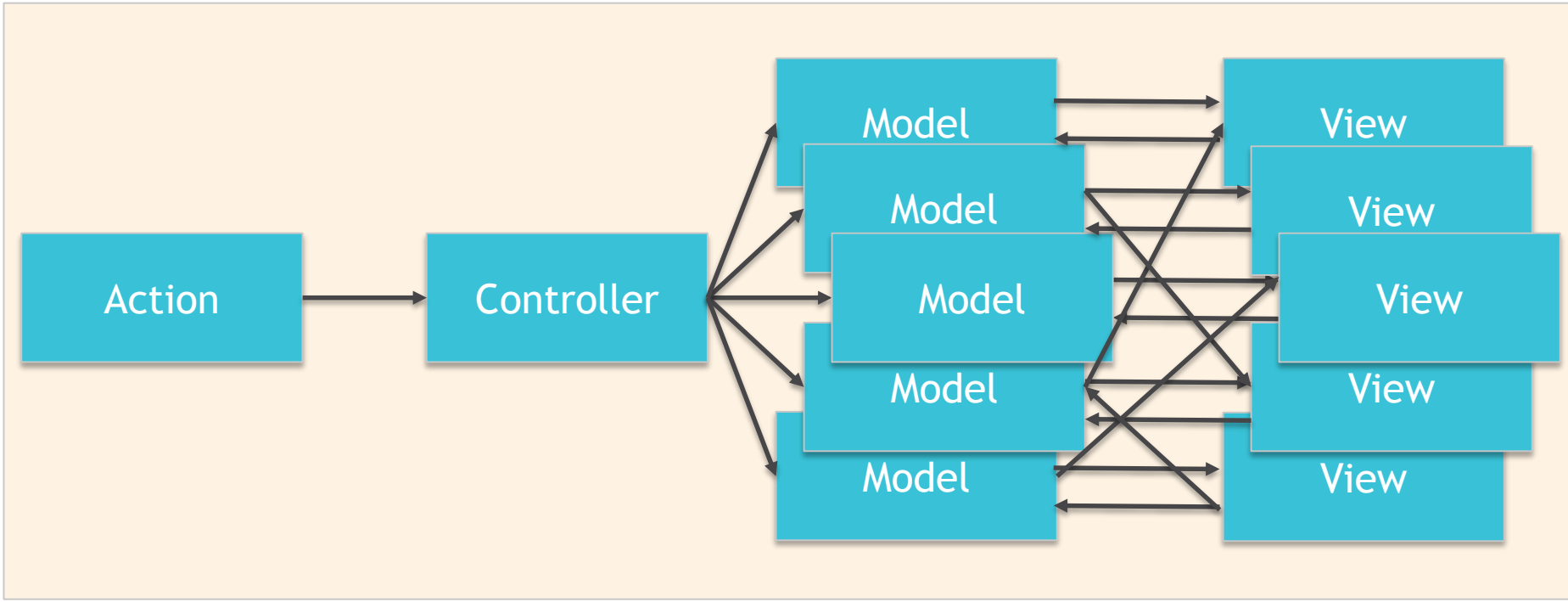
Flux base overview

What's wrong with MVC/MVVM?



Flux base overview

What's wrong with MVC/MVVM?



Flux base overview

What's wrong with MVC/MVVM?

- Good for small projects
- Doesn't have room for the new features
- Two-way data bindings led to cascading updates, where changing one object led to another object changing, which could also trigger more updates.
- Difficult to predict what would change as the result of one user interaction.

Flux structure

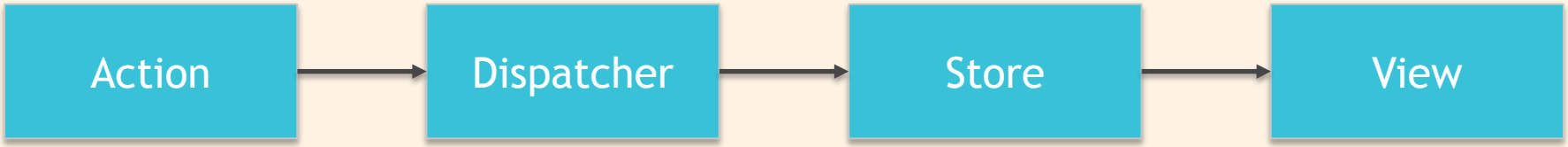
Flux Structure

What does Flux suggest?

- Dispatcher
- Action / Action creators
- Store
- View
- **Unidirectional data flow**

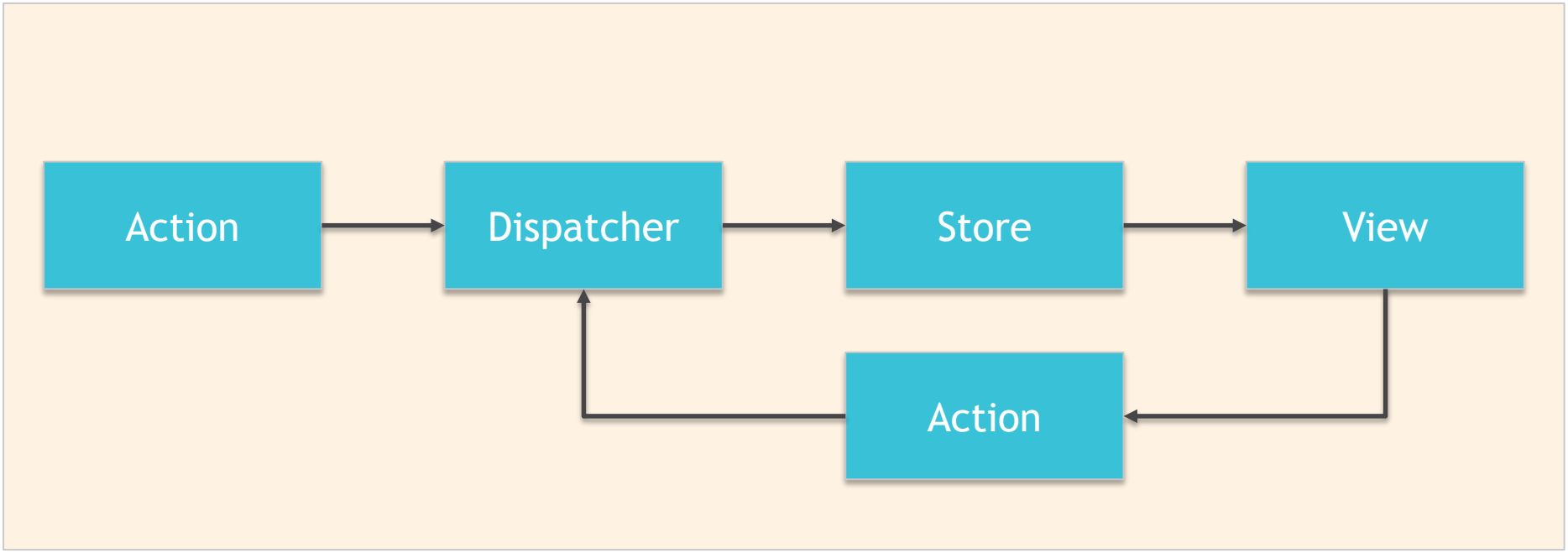
Flux Structure

How does it look like?



Flux Structure

How does it look like?



Flux Structure

Flux data flow

- All data flows through the dispatcher
- Dispatcher invokes callbacks, the store has registered with it
- Stores respond to relevant actions
- Stores emit a change event to registered controller views
- Controller views can listen to these changes via an event handler
- Controller views call render() via setState()
- Views may cause new actions

Flux Structure

What are the benefits

- Added predictability to the system
- Improved data consistency
- Less regression errors
- Easy to pinpoint root of a bug

Dispatcher

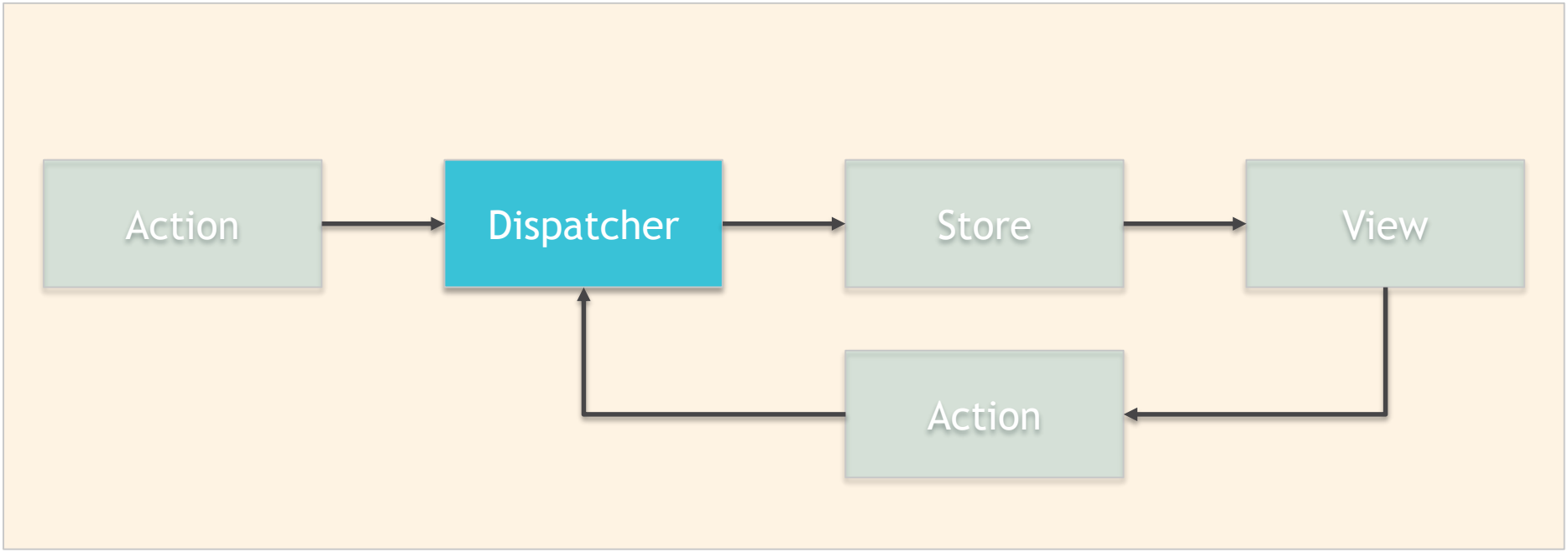
Dispatcher

What is **Dispatcher**?

- Singleton Object that acts as a central hub which manages all application data flow
- Distributes actions to stores
- Each store registers itself with a provided callback
- Until store layer is done, no other actions can come through the system (no cascading updates)

Dispatcher

In Flux scheme



Dispatcher

What is **Dispatcher**?

- Singleton Object that acts as a central hub which manages all application data flow
- Distributes actions to stores
- Each store registers itself with a provided callback
- Until store layer is done, no other actions can come through the system (no cascading updates)

Dispatcher

Simple API

- **register(function callback): string** Registers a callback to be invoked with every dispatched payload. Returns a token that can be used with waitFor().
- **dispatch(object payload): void** Dispatches a payload to all registered callbacks.
- **waitFor(array<string> ids): void** Waits for the callbacks specified to be invoked before continuing execution of the current callback. This method should only be used by a callback in response to a dispatched payload.
- **unregister(string id): void** Removes a callback based on its token.
- **isDispatching(): boolean** Is this Dispatcher currently dispatching.

Dispatcher

Each store registers in **Dispatcher**

```
Dispatcher.register(function (action) {
  switch (action.type) {
    case constants.ADD_TODO:
      TodoStore.addTodo(action.data);
      TodoStore.emit('change');
  }
});
```

Dispatcher

Each action dispatched to every registered callback

```
let addTodoAction = {
  type: 'ADD_TODO',
  data: {
    title: 'Do something'
  }
};

Dispatcher.dispatch(addTodoAction);
```

Actions/Action creators

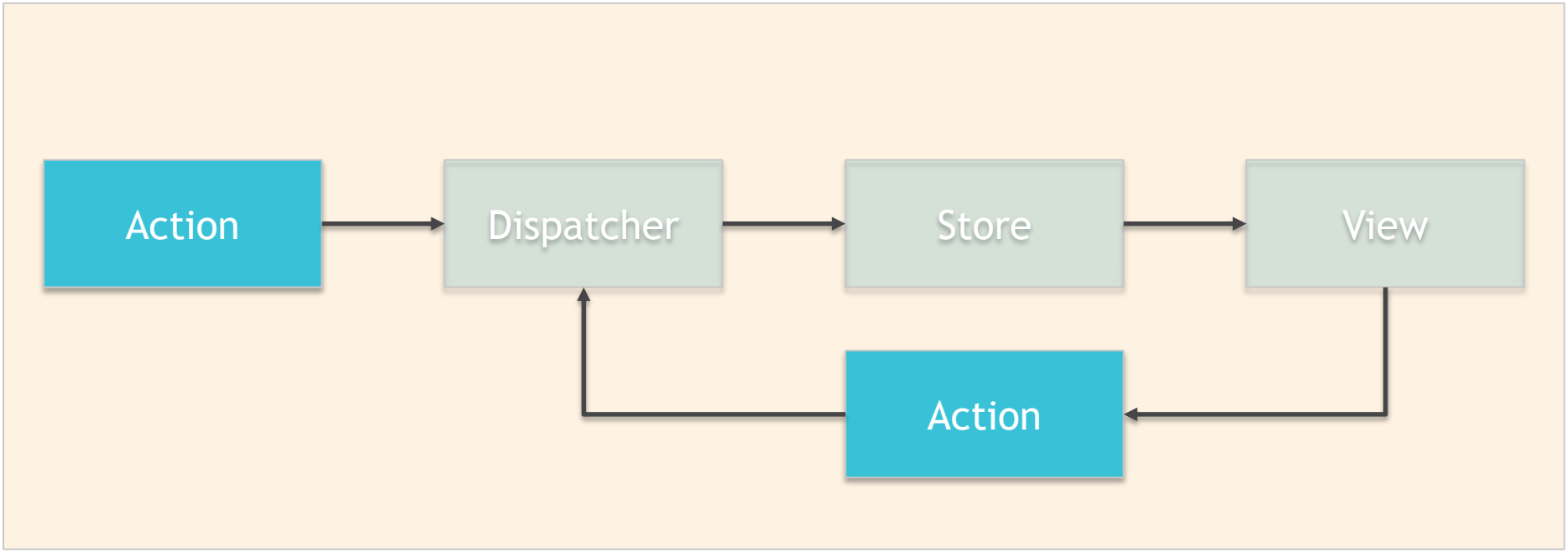
Actions

What are Actions?

- Is simple object containing the new data and an identifying *type* property.
- Necessary to inform that new data enters the system
- Necessary to inform that a person interacting with the application

Actions

In Flux scheme



Actions

Example

```
let addTodoAction = {
  type: 'ADD_TODO',
  data: {
    title: 'Do something'
  }
};

Dispatcher.dispatch(addTodoAction);
```

Actions

What is Action Creators?

- Provides a semantic API for work with logically grouped actions
- Facilitate passing data to the dispatcher
- Pass data to the dispatcher in form of an action

Actions

Example?

```
const TodoActions = {
  add(title) {
    Dispatcher.dispatch({
      type: 'ADD_TODO',
      data: {
        title: title
      }
    });
  },

  update(id, newTitle) {
    Dispatcher.dispatch({
      type: 'UPDATE_TODO',
      data: {
        id,
        title: newTitle
      }
    });
  }
};
```

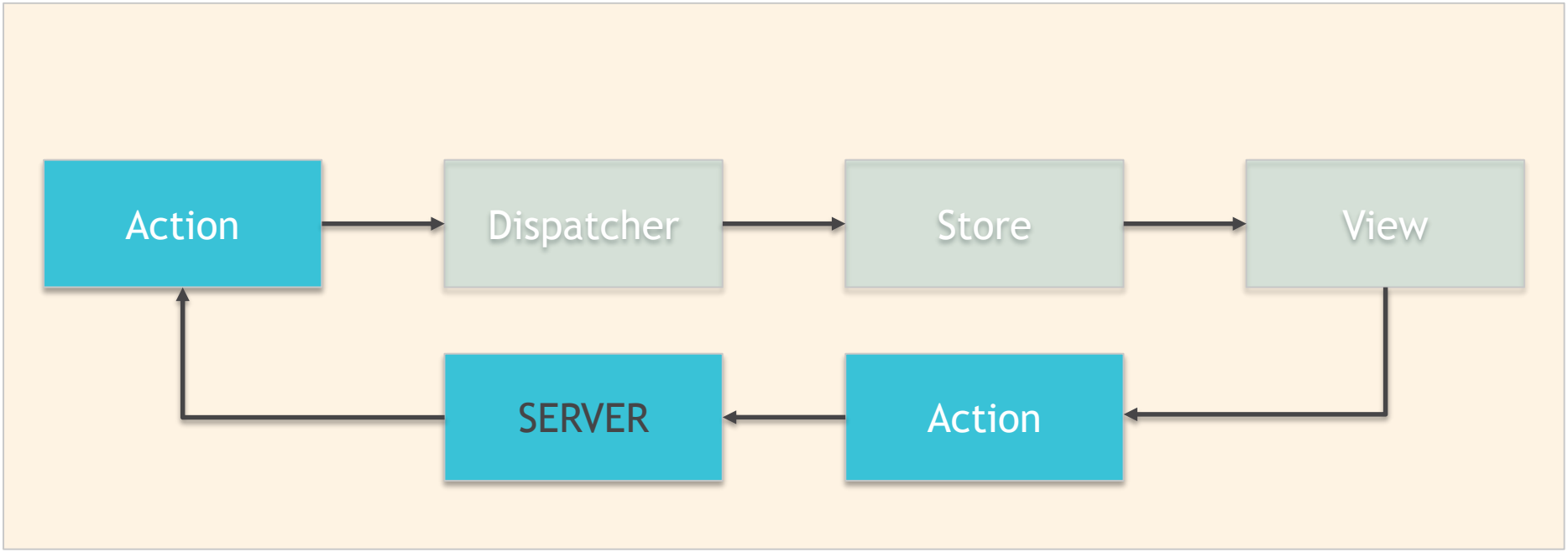
Actions

Server interactions

Hoe do we perform/handle server requests?

Actions

Server interactions



Actions

Example?

```
const TodoActions = {
  loadTodos() {
    Dispatcher.dispatch(constants.TODOS_LOADING);

    fetchTodos()
      .then((data)=> Dispatcher.dispatch({ type: constants.TODOS_FETCHED, data }))
      .catch((error)=> Dispatcher.dispatch({ type: constants.TODOS_FAILED, error }));
  }
};
```

Stores

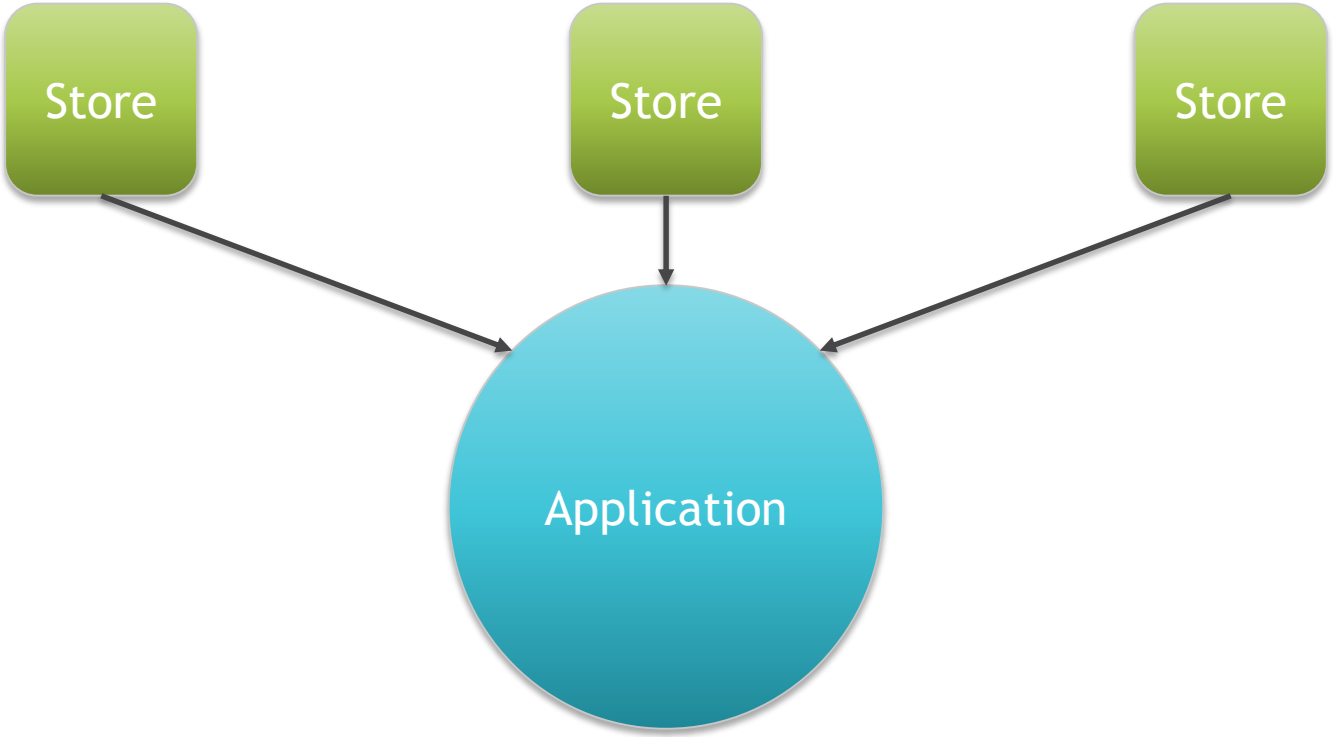
Stores

What is Stores?

- Domain model of the application (or part of app)
- Contain data and business logic
- Register with dispatcher
- Accept and handle updates (actions)
- Have no methods for mutations. Let’s consider them as read-only.

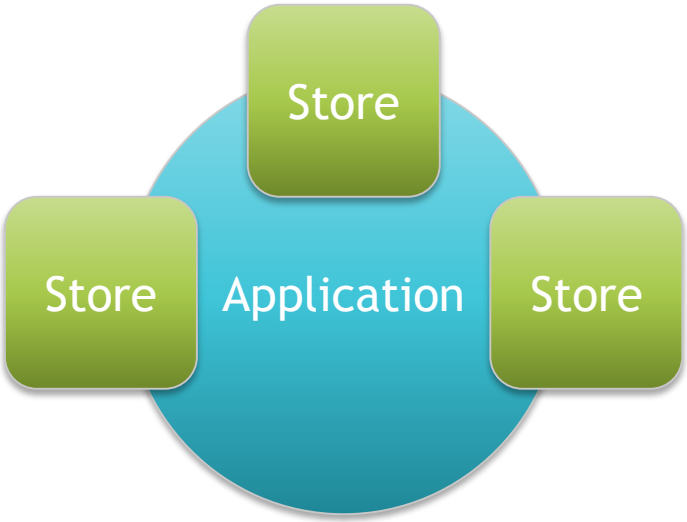
Stores

Application get data from the stores



Stores

Application set properties depending on stores state



And let this data flow to the sub-components

Actions

Communication with store

How views understand that stores were changed?

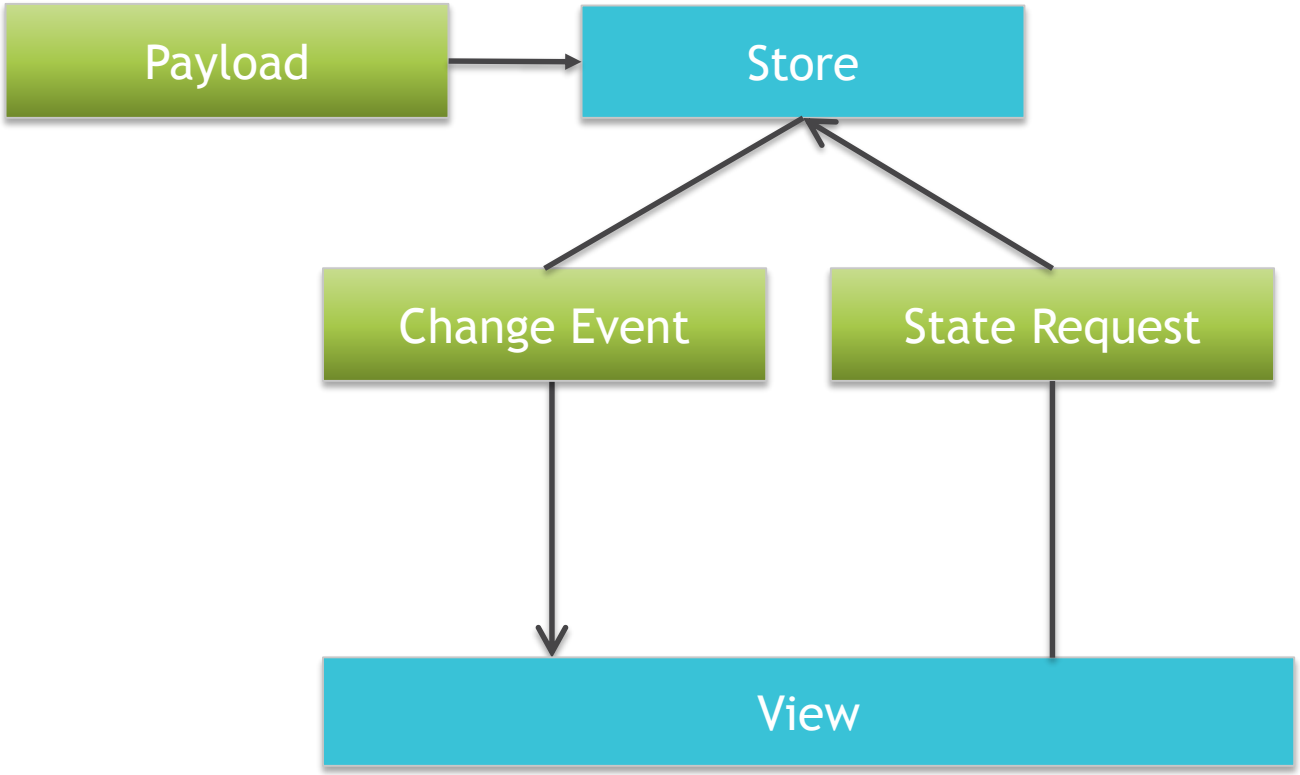
Stores

Communication with store

- Stores – are event emitters
- Emit ‘change’ event when their state changes.
- Views subscribe on store changes

Store

Communication with store



Stores

Example?

```
const TodoStore = {
  getTodos() {
    return _todos; // list of todos in state
  },

  subscribe(cb) {
    this.on('change', cb);
  },

  unsubscribe(cb) {
    this.off('change', cb);
  }
};

Dispatcher.register(function (action) {
  switch (action.type) {
    case constants.ADD_TODO:
      TodoStore.addTodo(action.data);
      TodoStore.emit('change');
  }
});
```

Store is event emiter and can emit events

Actions

Another way how to interact with server?

```
Dispatcher.register(function (action) {
  switch (action.type) {
    case constants.TODOS_FETCH:
      TodoStore.fetchTodos()
        .then((todos)=> {
          TodoStore.update(todos);
          TodoStore.emit('change');
        });
  }
});
```

Views

Views

What is View in FLUX?

- Better consider as View-Controllers or state-only or whatever.
- Register with stores to be notified of changes
- Receive state from the store
- Pass data from store to sub-components via props

Views

Example?

```
class Todo extends React.Component {
  constructor() {
    this.onChange = this.onChange.bind(this);
    this.state = {
      todos: TodoStore.getTodos()
    }
  }

  componentDidMount() {
    TodoStore.subscribe(this.onChange);
  }

  componentWillUnmount() {
    TodoStore.unsubscribe(this.onChange);
  }

  onChange() {
    this.setState({ todos: TodoStore.getTodos() });
  }

  render() {
    // logic of rendering sub components here...
  }
}
```

Actions

Who can trigger actions to invoke store changes?

Views

Actions

This should be handle in View Controllers and passed through callbacks to the child components

THANK YOU

QUESTIONS...