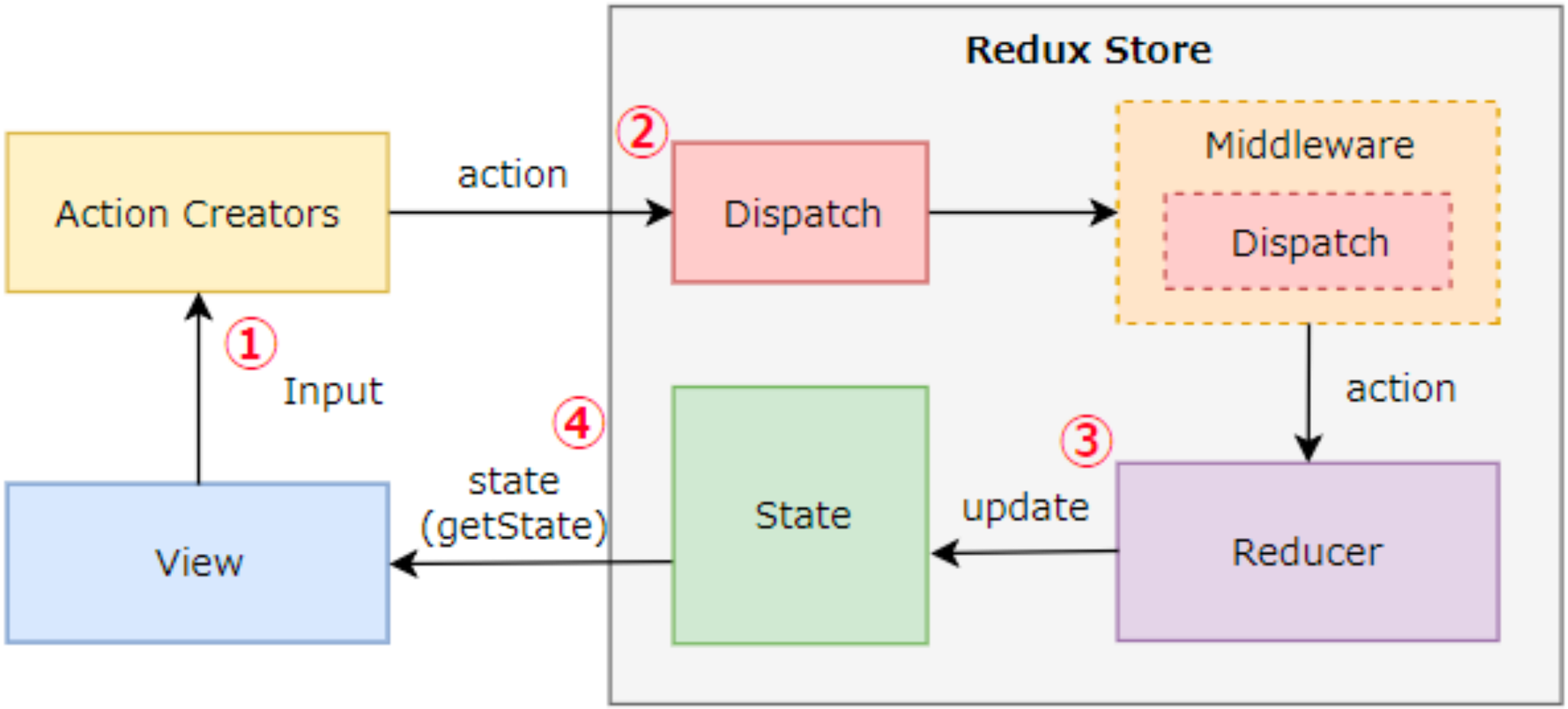
4つの要素 ーREDUXのみ



Store ： アプリケーションの状態とロジックを保持している場所

Reducer : Storeが保持している状態を変化させる関数

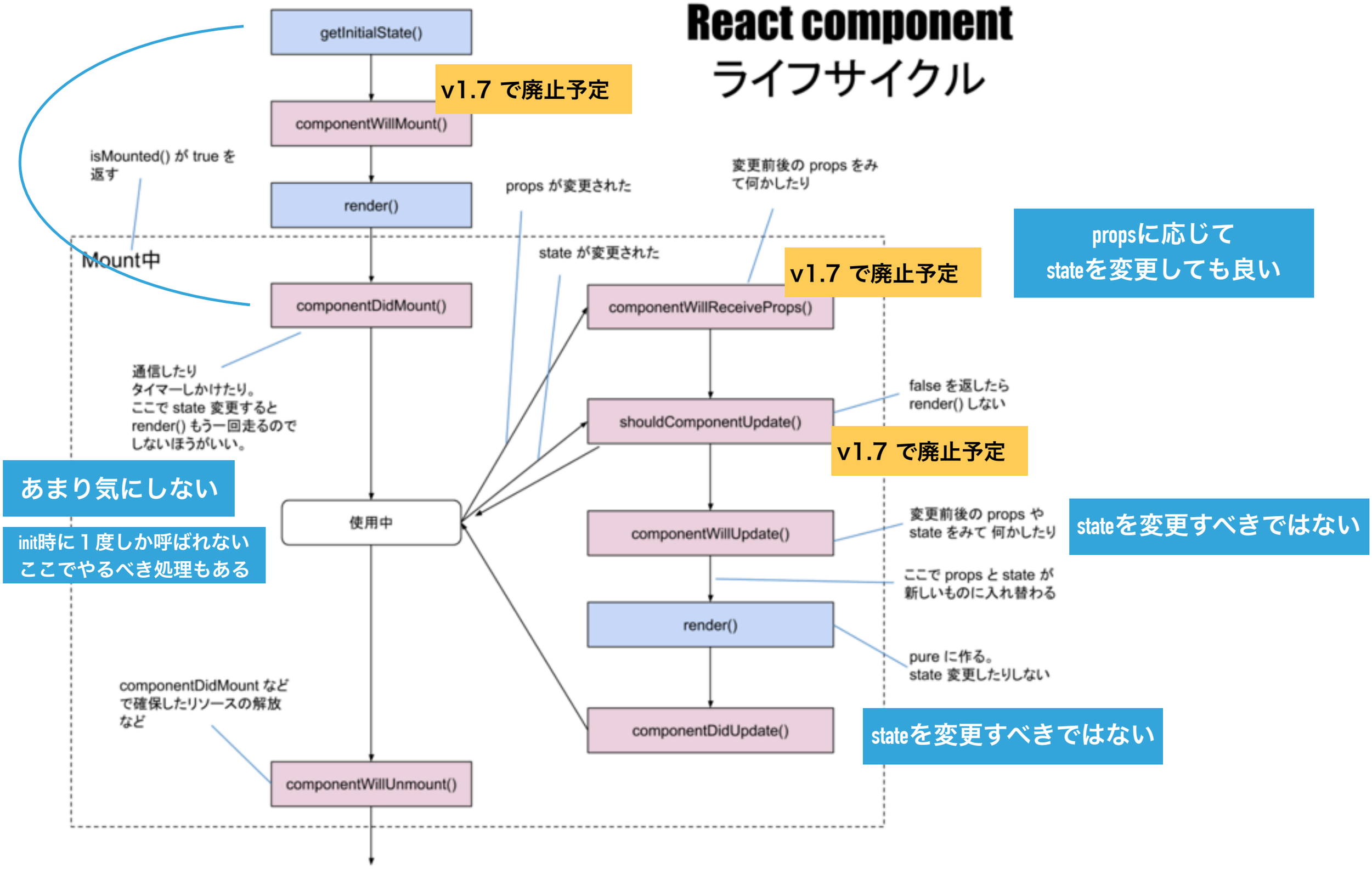
Action :　 状態変化を引き起こす現象（ユーザーの入力、APIからの情報取得）

Store

- dispatch　：actionをReducerに渡す
- subscribe：storeの状態が変化した時のコールバックを扱う
- getState　：storeの状態を参照できる
- replaceReducer：reducerを動的に割り当てる（使ったことない。。）

Reduxで管理する時、subscribeとgetState は明示的に呼び出すことは少ない。

ネットで拾ったもの


React component
ライフサイクル
getInitialState()
componentWillMount()
render()
componentDidMount()
使用中
componentWillUnmount()
componentWillReceiveProps()
shouldComponentUpdate()
componentWillUpdate()
render()
componentDidUpdate()
Mount中
isMounted() が true を返す
通信したり
タイマーしかけたり。
ここで state 変更すると
render() もう一回走るので
しないほうがいい。
componentDidMount など
で確保したリソースの解放
など
props が変更された
state が変更された
変更前後の props をみて何かしたり
false を返したら
render() しない
変更前後の props や
state をみて 何かしたり
ここで props と state が
新しいものに入れ替わる
pure に作る。
state 変更したりしない
v1.7 で廃止予定
v1.7 で廃止予定
v1.7 で廃止予定
あまり気にしない
init時に１度しか呼ばれない
ここでやるべき処理もある
propsに応じて
stateを変更しても良い
stateを変更すべきではない
stateを変更すべきではない

Spread Operator

```
...state,
```

```
return {
  <React.Fragment>
    <div></div>
    <div></div>
  </ React.Fragment>
}
```

```
return {
  <>
    <div></div>
    <div></div>
  </ >
}
```

Presentational Components

Redux異存のない純粋なReactのコンポーネント

Container Components

ReduxのStoreやActionを受け取り、Reactコンポーネントのpropsに渡す

Container

```
connect(

mapStateToProps,

mapDispatchToProps

)(Dakoku);
```

Componentに渡す（connect）

Component

```
Id = this.props.id;

onClick{(e) => this.props.addTODO()}
```

Storeから必要なStateを取り出し、Componentのpropsに渡す

```
const mapStateToProps = (state, ownProps) => ({
    id: state.Dakoku.id,
    messages: state.Dakoku.messages,

});
```

④のupdateされたstate

componentが持ってるstate

（古いpropsや、最初に設定した値や）

Component

```
export default class Dakoku extends React.Component{

    render(){
        const {id,year,month,messages,postDakoku,hh,mm,ss} = this.props;
    return(
        <div>
            <div>
            <p>id :{id}</p>
            </div>…
```

最初に宣言しなくても、

{this.props.id} のように参照できる

Storeのstateを変更する：dispatchをcomponentに渡す

component側は、直接stateを変更したり、setState()を呼んだりせずに

dispatchを呼び出すメソッドを受け取り、そのメソッドを呼ぶ

```
const mapDispatchToProps = dispatch => ({
   postDakoku(id,dakokuType) {
        dispatch(actions.postDakoku(id,dakokuType))
   }
});
```

Component

```
<button className=“dakoku_btn"
onClick= {() => postDakoku(id,"SYUKKIN")}> 出勤 </button>
```

打刻の例

API

localhost:8099/demo/jobs/${id}/${year}/${month}/${day}/

クライアント → サーバー

```
Url  :  localhost:8099/demo/jobs/1/2018/11/21

method  :  POST

body  : {
        ___time : 09:00:00,
        dakokuType : SYUKKIN
}
```

サーバー側

JobsController

```
@CrossOrigin
@RequestMapping(value = “/{id}/{year}/{month}/{day}",
                    method = RequestMethod.POST)
public Job createJob(
            @PathVariable("id") int id,
            @PathVariable("year") String year,
            @PathVariable("month") String month,
            @PathVariable("day") String day,
            @Validated @RequestBody JobInputBean input
           ) {

  Job insertJob=jobService.createJob(id,year,month,day,input);
  return insertJob;
}
```

テキスト

JobService

```
    public Job createJob(int id, String year, String month,
String day, JobInputBean input) {

        JobPk key = new JobPk();
        key.setId(id);

        StringBuilder sb = new StringBuilder(year);
        sb.append(DELIMITER_DATE);
        sb.append(month);
        sb.append(DELIMITER_DATE);
        sb.append(day);
        key.setDate(sb.toString());

        Optional<Job> jobOpt = jobsRepository.findById(key);

        Job job = new Job();
        if(jobOpt.isPresent()) {
            job = jobOpt.get();
            // json body からきた情報を追加
            setDakoku(input,job);
        } else {
            job.setId(id);
            job.setDate(sb.toString());
            setDakoku(input,job);
            // ...
        }

        return jobsRepository.save(job);
```

```
private void setDakoku(
        JobInputBean input, Job job) {

    switch (input.getDakokuType()) {
    case "SYUKKIN":
        job.setStartTime(input.getStartTime());
        break;

    case "TAIKIN":

        job.setEndTime(input.getEndTime());
       break;
    case “” ... ... :
        ... ...
}
```

流れ

1. {出勤} ボタンクリック

2.　日付取得　　: ex) 2018/11/23 08:50:00

3.　/demo/jobs/{id}/{year}/{month}/{day}　にPOST　する　→url とjson作成

```
{
  startTime : 08:50:00,
   dakokuType : SYUKKIN
}
```

テキスト

打刻漏れ、遅刻等、サーバーでわかるエラーは行（<tr>)を赤くする

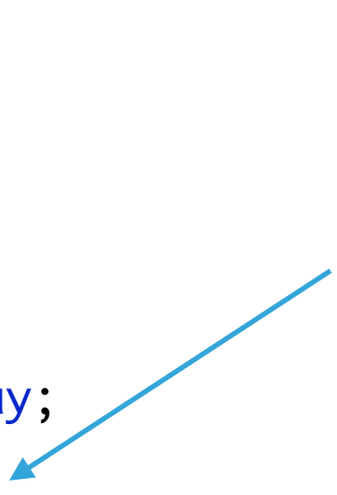
2018/11/02(金)			

クライアントサイドバリデーションは列（<td>)を赤くする

2018/11/06(火)	11:41:3 hh:MM:ss で入力してください	16:58:33	5.283333333333333
2018/11/07(水)	必須です		

Jobクラス（勤怠一覧の１行の情報）

```
    private Integer id;
    private String date;

    private String startTime;
    private String endTime;
    private String restStartTime;
    private String restEndTime;

     // 勤務日なら０、その他はまた考える
    private int jobStateCode;

    // 曜日（文字）
    private String dayOfWeek;
    // 1~7
    private String dayOfWeek;

    //  祝日
    private int holydayStatus;
    private String holydayName;

     // stab (使わないかも)
    private String workPerDay;
    private int restPerDay;
    private String overTimePerDay;

    private int dakokuErrorFlag;
    private List<String> errorMessages;
```

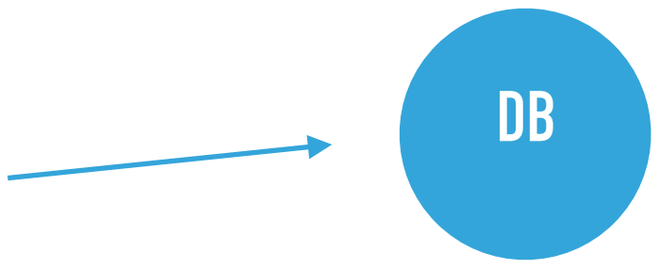


「打刻漏れ」「遅刻」等、
とりあえず区別せずに、
サーバー側でわかるエラー有無

表示する内容
（Listの１件目だけ）

サーバーサイドの人　ーー　打刻修正するAPIを作って

```
@RequestMapping(value = "/{id}/{year}/{month}", method = RequestMethod.POST
 public List<Job> correctJobDakoku(@PathVariable("id") int id,
                    @PathVariable("year")String year,  @PathVariable("month") String month,
                    @Validated @RequestBody List<Job> inputs)){

    List<Job> inputs;      // DBを更新or新規登録したいJobリスト

    Optional<Job> jobOpt = jobsRepository.findById(key);

    // DBにレコードがなかったら

       //  create
       Job = new Job()   => set~~
    // あったら

      Job = jobOpt.get()  => set~~
    }

    List<Job> updatedJobs
```



1.　Validationの結果 で送信ボタンを押せる/押せない

invalid ? button=>disabled : button => click OK

2.　startTime,endTime ~ に変更のあったものだけをPOST

始めにサーバーから受け取った値 : State.Jobs.kintais[1~29,30,31] .initial~Time

If (kintai[startTime] !== kintai[initialStartTime]) => Listか配列に

クライアントサイドの人　（validation）

```
const kintai = action.payload.kintais[action.payload.index];
    const properyName = action.payload.name;    // “startTime”, “endTime” とかカラム名
    kintai[properyName] = action.payload.value;
    kintai[properyName+'Validate'] = isValid;
    kintai[properyName+'Messages'] = message ? message : [];

    return action.payload.kintais.slice();
```

/reducer/Jobs.js がStoreのstateに保存する、kintais[]　をそのまま使ってます。

kintai[properyName] -> kintai.startTIme と同じ

kintai.startTime = action.payload.value // (event.target.value : inputのvalue　を受け取ってる）