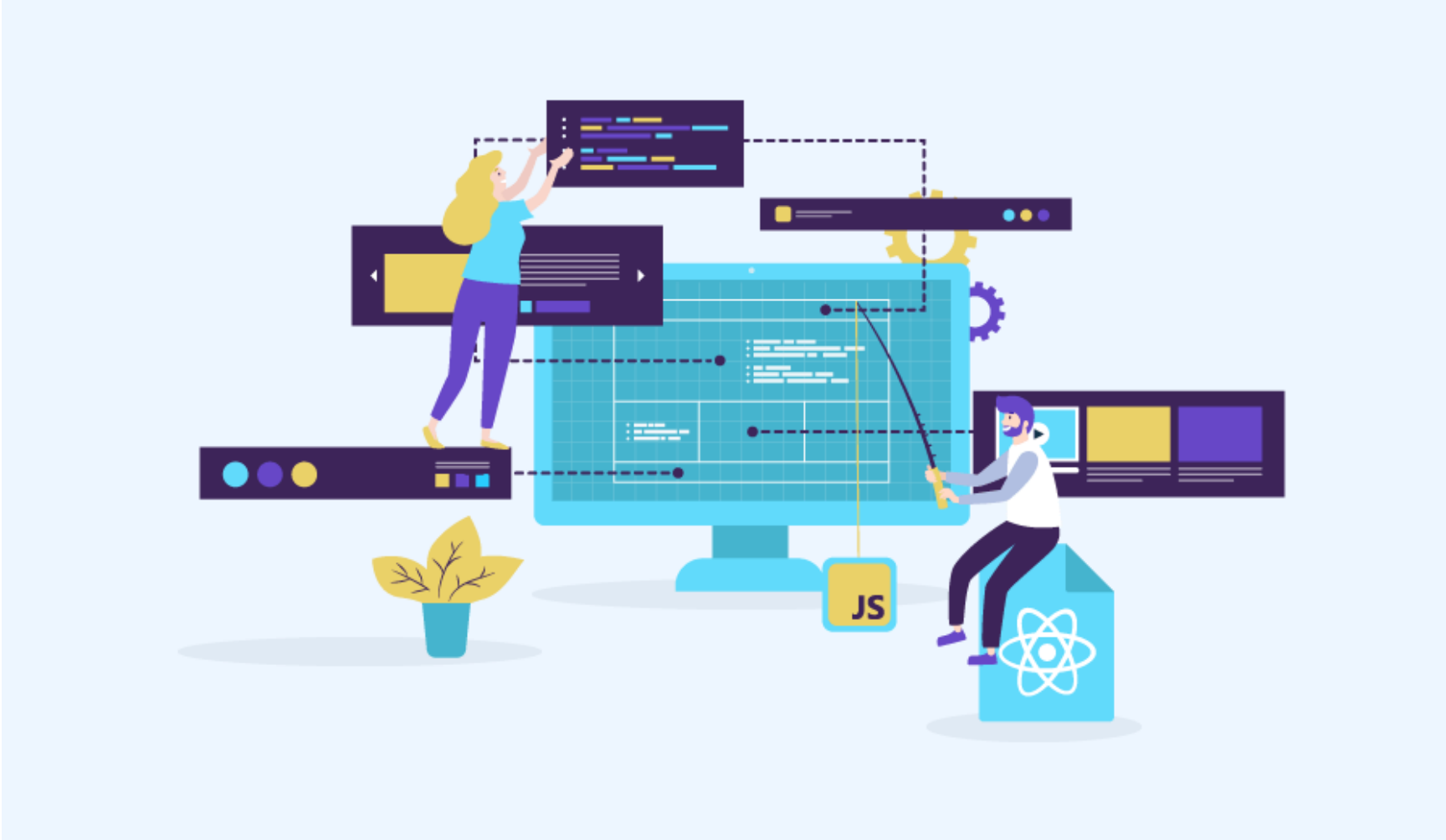
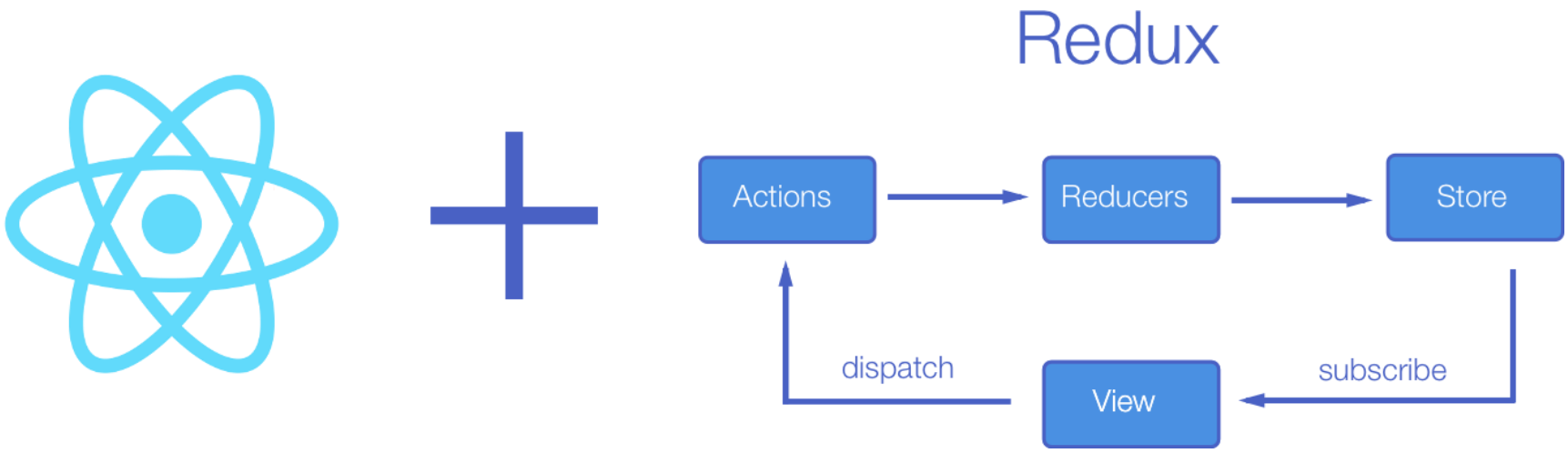
Treinamento em REACT

Professor: Sergio Mendes

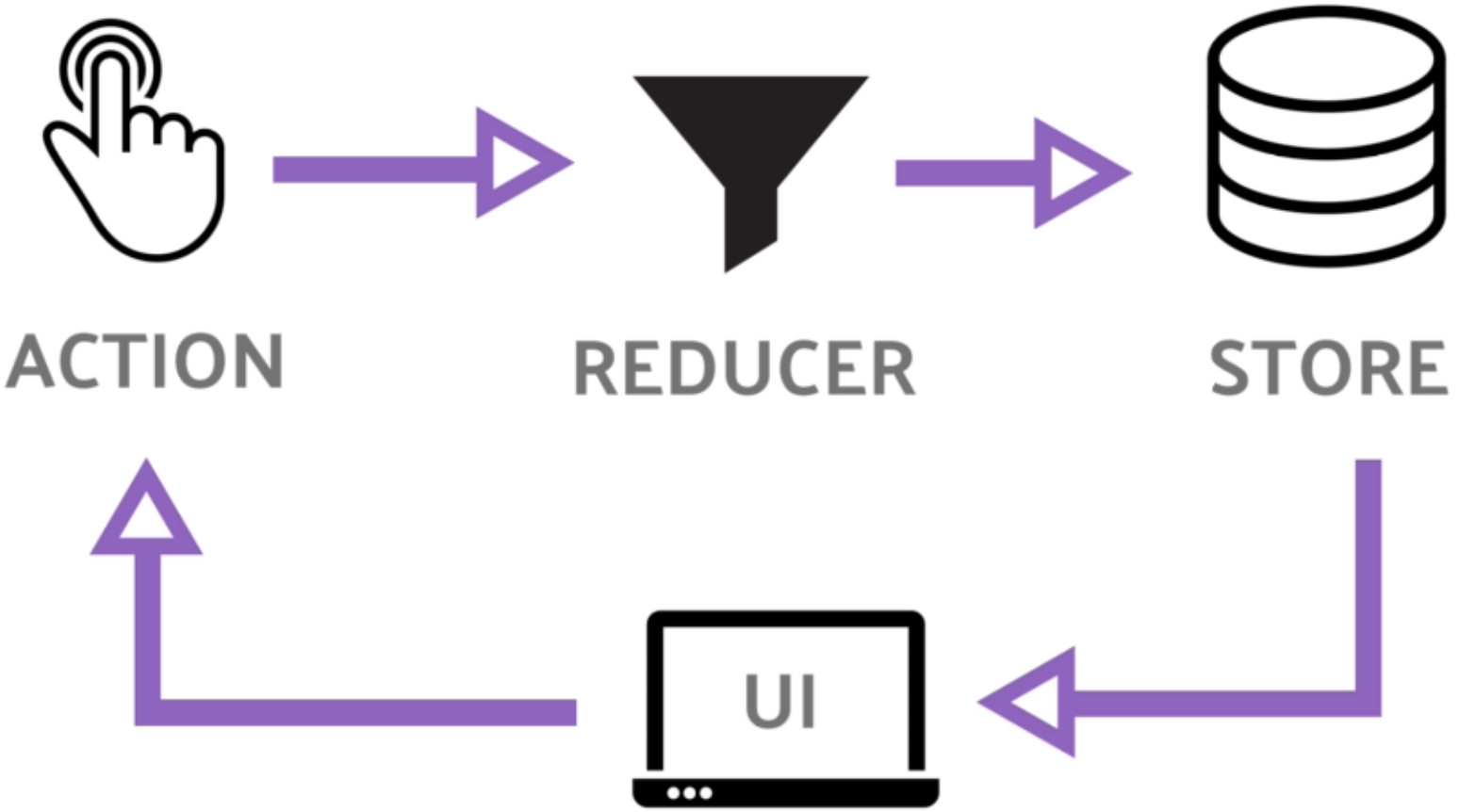


Redux

O Redux simplifica a evolução de estados de uma aplicação quando há múltiplos estados para controlar e muitos componentes que precisam atualizar ou se inscrever nessa evolução, tirando a responsabilidade de cada componente de guardar o estado e passando para uma centralizada e única **Store**.



Fluxo de evolução de estado



Store:
é o container que armazena e centraliza o estado geral da aplicação. Ela é imutável, ou seja, nunca se altera, apenas evolui.

Actions:
são fontes de informações que são enviadas da aplicação para a Store. São disparadas pelas **Action Creators**, que são simples funções que, ao serem executadas, ativam os Reducers.

Reducers:
recebem e tratam as informações para que sejam ou não enviadas à Store.

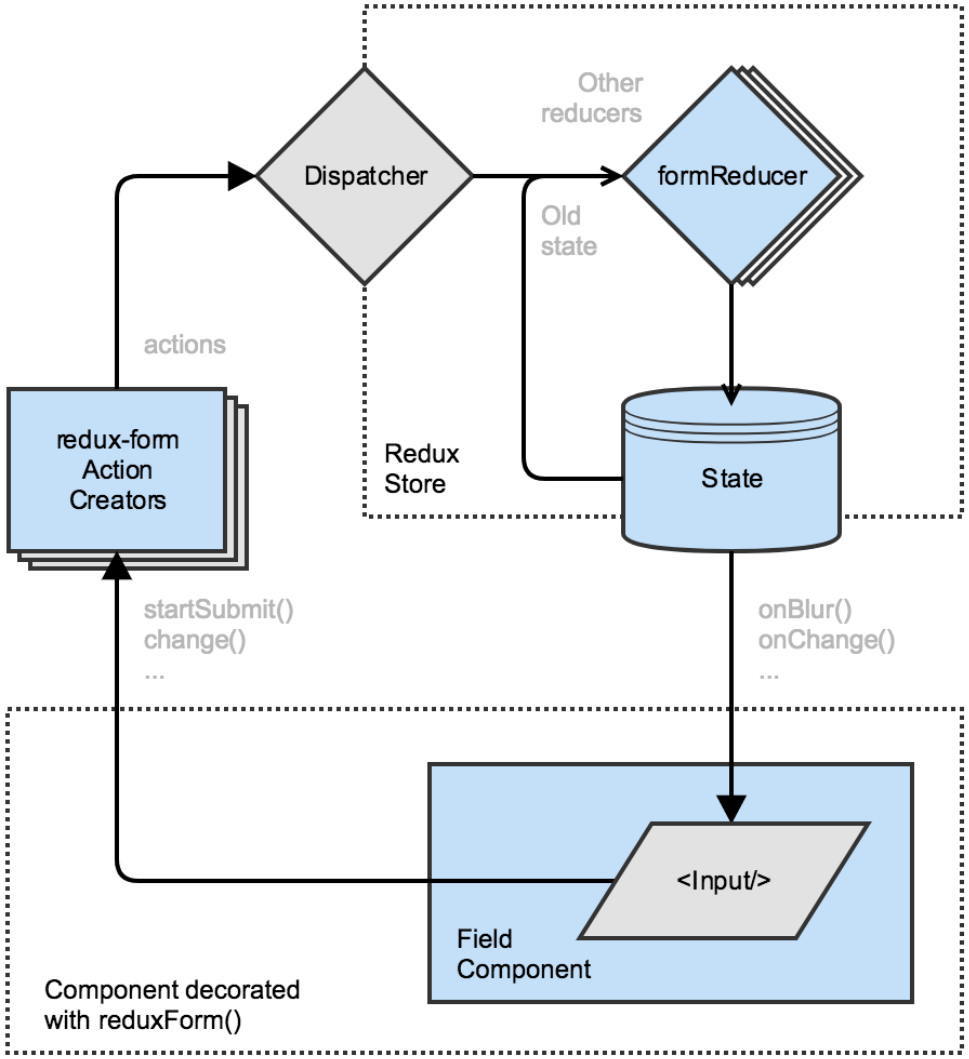
Conexão dos componentes ao Redux:
para poderem se inscrever à evolução de estados da Store ou disparar eventos para evoluí-la.

Redux Forms

Biblioteca React para desenvolvimento de formulários através do Redux.
É capaz de integrar as ações de formulário ao state da aplicação, incluir recursos de validação, etc.

Treinamento em REACT

Professor: Sergio Mendes



Treinamento em REACT

Professor: Sergio Mendes

Bibliotecas:

axios:

biblioteca que auxilia em requisições HTTP;

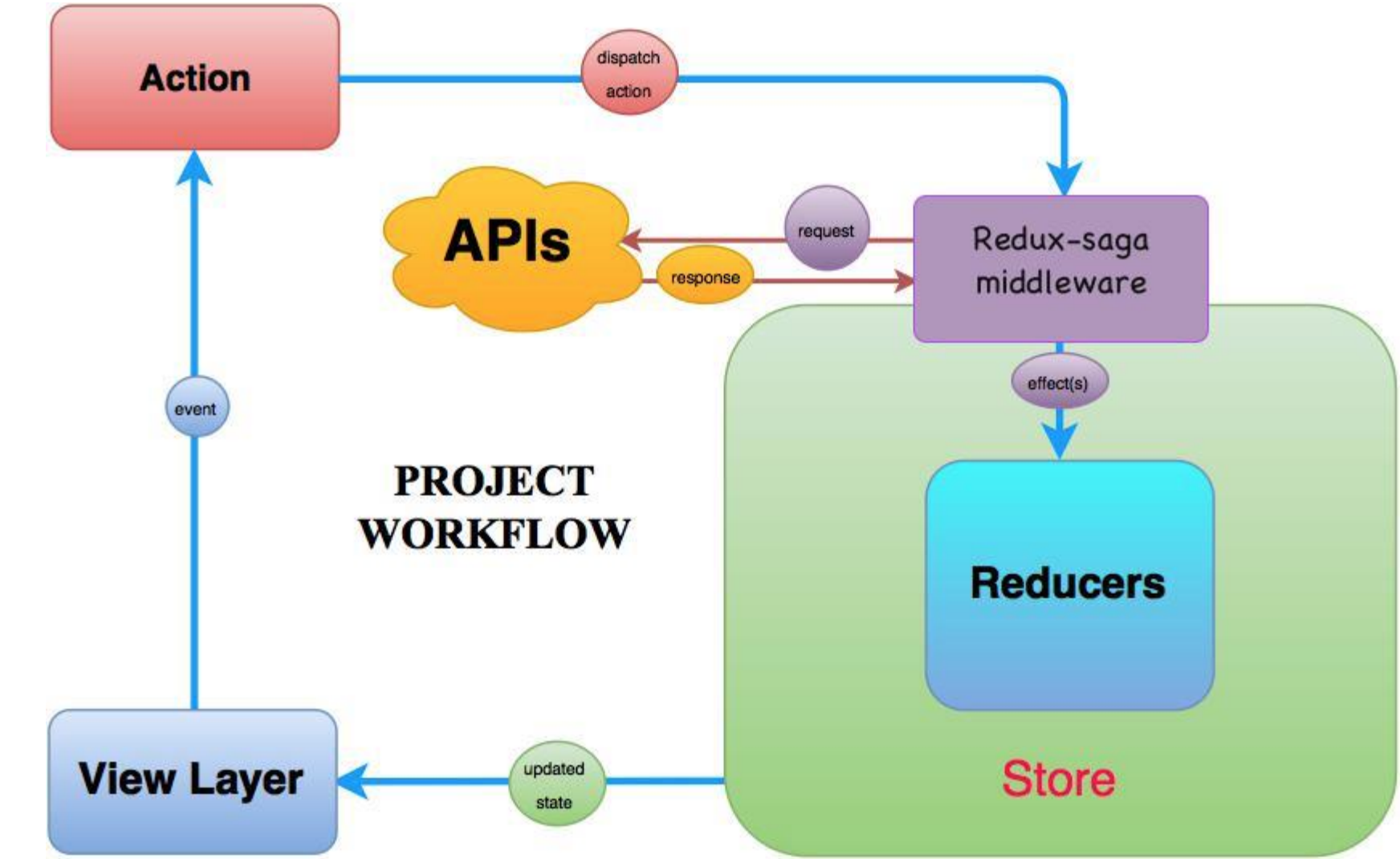

```
const apiUrl = 'http://localhost:4000/posts';

export const createPost = ({ title, body }) => {
  return (dispatch) => {
    return axios.post(`${apiUrl}/add`, {title, body})
      .then(response => {
        dispatch(createPostSuccess(response.data))
      })
      .catch(error => {
        throw(error);
      });
  };
};
```

Redux Saga

Redux-saga é uma biblioteca que foca em fazer os efeitos colaterais (ex: chamadas assíncronas para buscar dados em uma API, acessar o cache do navegador, etc) em aplicações React/Redux serem mais fáceis e simples de se criar e manter.

Treinamento em REACT

Professor: Sergio Mendes



Redux Saga expõe vários métodos chamados de **Effects**, e vamos usar vários deles:

fork(), realiza uma operação não bloqueante com a função passada
take(), pausa as operações até receber uma **redux action**
race(), executa **Effects** simultaneamente, e cancela todos quando um efeito retorna seu resultado
call(), executa uma função. Se essa função retornar uma Promise, ele irá pausar a Saga até a Promise ser resolvida
put(), despacha uma **redux action**
select(), executa uma função seletora que irá buscar dados do estado global do Redux
takeLatest(), irá executar as operações recebidas, porém, irá retornar apenas o valor da última. Se a mesma operação for enviada mais de uma vez, elas serão ignoradas, exceto a última (ex: click -> loadUser, usuário clica 4 vezes no botão (ele é legal né, quer testar sua app), apenas a função enviada no último click será executada/retornado o valor, as outras serão ignoradas)
takeEvery(), irá retornar os valores de todas as operações recebidas