

Presentation by Uplatz

Contact Us: <https://training.uplatz.com/>

Email: info@uplatz.com

Phone: +44 7836 212635

Table Of Contents:

- Flux
- Redux
- Redux Middleware
- Redux actions

What is flux

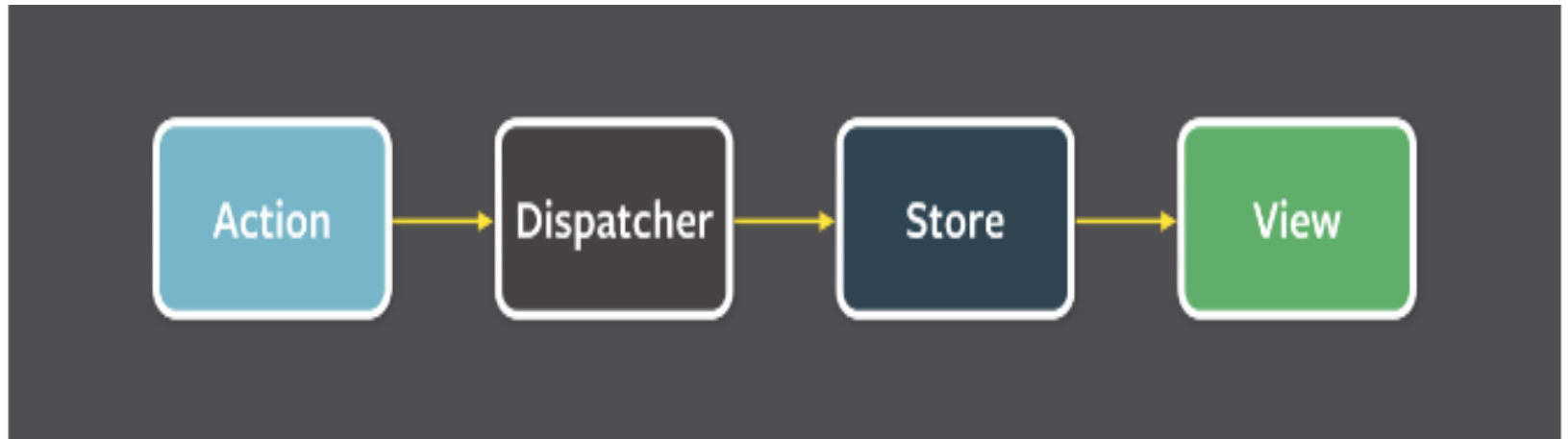
- Flux is a pattern for managing how data flows through a React application.
- As we've seen, the preferred method of working with React components is through passing data from one parent component to its children components.
- The Flux pattern makes this model the default method for handling data.
- There are three distinct roles for dealing with data in the flux methodology:

Dispatcher

Stores

Views (our components)

- The major idea behind Flux is that there is a single-source of truth (the stores) and they can only be updated by triggering actions.
- The actions are responsible for calling the dispatcher, which the stores can subscribe for changes and update their own data accordingly.
- When a dispatch has been triggered, and the store updates, it will emit a change event which the views can rerender accordingly.



- This may seem unnecessarily complex, but the structure makes it incredibly easy to reason about where our data is coming from, what causes it's changes, how it changes, and lets us track specific user flows, etc.
- The key idea behind Flux is:
- Data flows in one direction and kept entirely in the stores.

Implementations

- Although we can create our own flux implementation, many have already created some fantastic libraries we can pick from.
- Facebook's flux
- alt nuclear-js
- Fluxible

- reflux
- Fluxxor
- flux-react
- And more... many many more
- Plug for fullstackreact
- We discuss this material in-depth about Flux, using libraries, and even implementing our own version of flux that suits us best. It can be pretty intense trying to pick the right choice for our applications.
- Each has their own features and are great for different reasons.
- However, to a large extent, the React community has focused in on using another flux tool called Redux.

- It can be pretty intense trying to pick the right choice for our applications.
- Each has their own features and are great for different reasons.
- However, to a large extent, the React community has focused in on using another flux tool called Redux.


Redux

- Redux is a small-ish library that takes it's design inspiration from the Flux pattern, but is not itself a pure flux implementation.
- It provides the same general principles around how to update the data in our application, but in slightly different way.

- Unlike Flux, Redux does not use a dispatcher, but instead it uses pure functions to define data mutating functions.
- It still uses stores and actions, which can be tied directly to React components.
- The 3 major principles of Redux we'll keep in mind as we implement Redux in our app are:

Updates are made with pure functions (in reducers)

state is a read-only property

- state is the single source of truth (there is only one store in a Redux app)
- One big difference with Redux and Flux is the concept of middleware.
- Redux added the idea of middleware that we can use to manipulate actions as we receive them, both coming in and heading out of our application. 

Data Management with Redux

- The app we're building with it right now is bare-bones simple, which will just show us the last time the page fetched the current time.
- For simplicity for now, we won't call out to a remote server, just using the JavaScript Date object.
- The first thing we'll have to do to use Redux is install the library.
- We can use the npm package manager to install redux. In the root directory of our app we previously built, let's run the npm install command to install redux:

npm install --save redux

- We'll also need to install another package that we'll use with redux, the react-redux that will help us tie together react and redux:

npm install --save react-redux

```
auser@30days $ npm install --save redux react-redux
30days@0.0.1 /Users/auser/Development/javascript/mine/sample-apps/30days/30days
├─┬ react-redux@4.4.5
│   └── lodash@4.15.0
├─┬ redux@3.6.0
│   └── lodash-es@4.15.0
└── symbol-observable@1.0.2
```

```
auser@30days $
```

Configuration and setup

- The next bit of work we need to do is to set up Redux inside of our app.
- We'll need to do the following to get it set up:

Define reducers

Create a store

Create action creators

Tie the store to our React views

Profit

Precursor

- We'll talk terminology as we go, so take this setup discussion lightly (implementing is more important to get our fingers moving).
- We'll restructure our app just slightly (annoying, I know... but this is the last time) so we can create a wrapper component to provide data down through our app.

When we're complete, our app tree will have the following shape:

[Root] -> [App] -> [Router/Routes] -> [Component]

Without delaying any longer, let's move our src/App.js

- into the src/containers directory and we'll need to update some of the paths from our imports at the same time.
- We'll be using the react router material we discussed a few days ago.
- We'll include a few routes with the <Switch /> statement to ensure only one shows up at a time.

```
import React from "react";
```

```
import { BrowserRouter as Router, Route, Switch } from  
"react-router-dom";
```

```
// We'll load our views from the `src/views`  
// directory
```

```
import Home from "../views/Home/Home";
```

```
import About from "../views/About/About";
```

```
const App = props => {  
  return (  
    <Router>  
      <Switch>  
        <Route path="/about" component={About} />  
        <Route path="*" component={Home} />  
      </Switch>  
    </Router>  
  );  
};
```

```
export default App;
```

- In addition, we'll need to create a new container we'll call Root which will wrap our entire <App /> component and make the store available to the rest of the app.

Let's create the src/containers/Root.js file:

`touch src/containers/Root.js`

- For the time being, we'll use a placeholder component here, but we'll replace this content as we talk about the store.
- For now, let's export something:

```
import React from "react";
```

```
import App from "../App";
```

```
const Root = props => {  
  return <App />;
```

```
};
```

```
export default Root;
```

- Finally, let's update the route that we render our app in the src/index.js file to use our new Root container instead of the App it previously used.

```
import React from "react";  
import ReactDOM from "react-dom";  
import Root from "../containers/Root";  
import "../index.css";  
ReactDOM.render(<Root />,  
document.getElementById("root"));
```

Adding in Redux

- Now with a solid app structure in place, we can start to add in Redux.
- The steps we'll take to tie in some Redux structure are generally all the same for most every application we'll build. We'll need to:

Write a root reducer

Write actionCreators

- Configure the store with the rootReducer, the store, and the app

Connect the views to the actionCreators

- We'll purposefully be keeping this high-level introduction a tad short, so hang tight if that's a mouthful, it will all make more sense shortly.
- Let's setup the structure to allow us to add redux. We'll do almost all of our work in a src/redux directory.
- Let's create that directory.

mkdir -p src/redux

touch src/redux/configureStore.js

touch src/redux/reducers.js

Let's start by creating our reducer first.

- Although it sounds complex, a reducer is actually pretty straight-forward with some experience.
- A reducer is literally only a function.
- It's sole responsibility is to return a representation of the next state.
- In the Redux pattern, unlike flux we are only handling one global store for the entire application.
- This makes things much easier to deal with as there's a single place for the data of our application to live.
- The root reducer function is responsible to return a representation of the current global state of the application.
- When we dispatch an action on the store, this reducer function will be called with the current state of the application and the action that causes the state to update.

Let's build our root reducer in a file at
src/redux/reducers.js.

```
// Initial (starting) state
```

```
export const initialState = {  
  currentTime: new Date().toString()  
};
```

```
// Our root reducer starts with the initial state
```

```
// and must return a representation of the next state
```

```
export const rootReducer = (state = initialState, action)  
=> {  
  return state;  
};
```

In the function, we're defining the first argument to
start out as the initial state (the first time it runs, the

- rootReducer is called with no arguments, so it will always return the initialState on the first run).
- In the function, we're defining the first argument to start out as the initial state (the first time it runs, the rootReducer is called with no arguments, so it will always return the initialState on the first run).
- That's the rootReducer for now.
- As it stands right now, the state always will be the same value as the initialState.
- In our case, this means our data tree has a single key of currentTime.

What is an action?

- The second argument here is the action that gets dispatched from the store.

- We'll come back to what that means exactly shortly. For now, let's look at the action.
- At the very minimum, an action must include a type key.
- The type key can be any value we want, but it must be present.
- For instance, in our application, we'll occasionally dispatch an action that we want to tell the store to get the new current time.

We might call this action a string value of **FETCH_NEW_TIME**.

- The action we might dispatch from our store to handle this update looks like:

```
{  
  type: "FETCH_NEW_TIME"  
}
```

- As we'll be typing this string a lot and we want to avoid a possible misspelling somewhere, it's common to create a `types.js` file that exports the action types as constants.

Let's follow this convention and create a `src/redux/types.js` file:

`export const FETCH_NEW_TIME = "FETCH_NEW_TIME";`

- Instead of calling the action with the hard-coded string of `'FETCH_NEW_TIME'`, we'll reference it from the `types.js` file:

`import * as types from './types';`

```
{  
  type: types.FETCH_NEW_TIME,  
}
```

- When we want to send data along with our action, we can add any keys we want to our action.
- We'll commonly see this called payload, but it can be called anything.
- It's a convention to call additional information the payload.
- Our **FETCH_NEW_TIME** action will send a payload with the new current time.
- Since we want to send a serializable value with our actions, we'll send the string value of the new current time.{

```
type: types.FETCH_NEW_TIME,  
payload: new Date().toString() // Any serializable  
value  
}
```

- Back in our reducer, we can check for the action type and take the appropriate steps to create the next state.
- In our case, we'll just store the payload. If the type of the action is **FETCH_NEW_TIME**, we'll return the new currentTime (from our action payload) and the rest of the state (using the ES6 spread syntax):

```
export const rootReducer = (state = initialState, action)  
=> {  
  switch (action.type) {  
    case types.FETCH_NEW_TIME:  
      return { ...state, currentTime: action.payload };  
    default:  
      return state;  
  } };
```

- Remember, the reducers must return a state, so in the default case, make sure to return the current state at the very minimum.

Keep it light

- Since the reducer functions run everytime an action is dispatched, we want to make sure these functions are as simple and fast as possible.
- We don't want them to cause any side-effects or have much delay at all.
- We'll handle our side-effects outside of the reducer in the action creators.
- Before we look at action creators (and why we call them action creators), let's hook up our store to our application.

We'll be using the react-redux package to connect our views to our redux store.

Let's make sure to install this package using npm:

```
npm install --save react-redux
```

Hooking up the store to the view

- The react-redux package exports a component called Provider.
- The Provider component makes the store available to all of our container components in our application without needing for us to need to pass it in manually every time.
- The Provider component expects a store prop that it expects to be a valid redux store, so we'll need to complete a configureStore function before our app will run without error.

- For now, let's hook up the Provider component in our app.
- We'll do this by updating our wrapper Root component we previously created to use the Provider component.

```
import { Provider } from "react-redux";
```

```
// ...
```

```
const Root = props => {
```

```
  // ...return (
```

```
    <Provider store={store}>
```

```
      <App />
```

```
    </Provider>
```

```
  );
```

```
};
```

- Notice we're sending in the store value to our Provider component... but we haven't created the store yet! Let's fix that now.

Configuring the store

- In order to create a store, we'll use the new `src/redux/configureStore.js` to export a function which will be responsible for creating the store.

How do we create a store?

- The `redux` package exports a function called `createStore` which will create the actual store for us, so let's open up the `src/redux/configureStore.js` file and export a function (we'll define shortly) called `configureStore()` and import the `createStore` helper:

```
import { createStore } from "redux";
```

```
// ...
```

```
export const configureStore = () => {
```

```
  // ...
```

```
};
```

```
// ...
```

```
export default configureStore;
```

- We don't actually return anything in our store quite yet, so let's actually create the redux store using the createStore function we imported from redux:

```
import { createStore } from "redux";
```

```
export const configureStore = () => {
```

```
  const store = createStore();
```

```
  return store;
```


```
};
```

```
export default configureStore;
```

- Now let's update our Root.js file with an instance of the store created by calling the configureStore() function.

```
// ...
```

```
import configureStore from "../redux/configureStore";  
const Root = props => {  
  const store = configureStore();  
  return (  
    <Provider store={store}>  
      <App />  
    </Provider>  
  );  
};
```

- If we load our page in the browser, we'll see we have one giant error and no page gets render 

- The error redux is giving us is telling us that we don't have a reducer inside our store.
- Without a reducer, it won't know what to do with actions or how to create the state, etc.
- In order to move beyond this error, we'll need to reference our rootReducer we created.
- The createStore function expects us to pass the rootReducer in as the first argument.
- It'll also expect the initial state to be passed in as the second argument.
- We'll import both of these values from the reducers.js file we created.

```
import { rootReducer, initialState } from "../reducers";  
// ...
```

```
export const configureStore = () => {  
  const store = createStore(  
    rootReducer, // root reducer  
    initialState // our initialState  
  );  
  return store;  
};
```

Connecting the view (cont'd)

- Everything in our app is set-up to use Redux without too much overhead.
- One more convenience that redux offers is a way to bind pieces of the state tree to different components using the `connect()` function exported by the `react-redux` package.
- The `connect()` function returns a function that expects the 1st argument to be that of a component.

- This is often called a higher-order component.
- The `connect()` function expects us to pass in at least one argument to the function (but often we'll pass in two).
- The first argument it expects is a function that will get called with the state and expects an object in return that connects data to the view.
- Let's see if we can demystify this behavior in code.
- We'll call this function the `mapStateToProps` function.
- Since its responsibility is to map the state to an object which is merged with the component's original props.
- Let's create the Home view in `src/views/Home.js` and use this `connect()` function to bind the value of `currentTime` in our state tree.


```
import { connect } from "react-redux";
```

```
// ...
```

```
const mapStateToProps = state => {
```

```
  return {
```

```
    currentTime: state.currentTime
```

```
  };
```

```
};
```

```
export default connect(mapStateToProps)(Home);
```

- This `connect()` function automatically passes any of the keys in the function's first argument as props to the Home component.
- In our demo's case, the `currentTime` prop in the Home component will be mapped to the state tree key at `currentTime`.

- Let's update the Home component to show the value in the currentTime:

```
const Home = props => {  
  return (  
    <div className="home">  
      <h1>Welcome home!</h1>  
      <p>Current time: {props.currentTime}</p>  
    </div>  
  );  
};
```

- Although this demo isn't very interesting, it shows we have our Redux app set up with our data committed to the global state and our view components mapping the data.

Triggering updates(Redux)

- Recall that the only way we can change data in Redux is through an action creator.
- We created a redux store yesterday, but we haven't created a way for us to update the store.
- What we want is the ability for our users to update the time by clicking on a button.
- In order to add this functionality, we'll have to take a few steps:

Create an **actionCreator** to dispatch the action on our store

- Call the **actionCreator** onClick of an element
- Handle the action in the reducer
- We already implemented the third step, so we only have two things to do to get this functionality working as we expect.

- Yesterday, we discussed what actions are, but not really why we are using this thing called actionCreators or what they are.
- As a refresher, an action is a simple object that must include a type value.
- We created a types.js file that holds on to action type constants, so we can use these values as the type property.

```
export const FETCH_NEW_TIME = 'FETCH_NEW_TIME';
```

```
export const LOGIN = 'USER_LOGIN';
```

```
export const LOGOUT = 'USER_LOGOUT';
```

- As a quick review, our actions can be any object value that has the type key.

- We can send data along with our action (conventionally, we'll pass extra data along as the payload of an action).

```
{  
  type: types.FETCH_NEW_TIME,  
  payload: new Date().toString()  
}
```

- Now we need to dispatch this along our store.
- One way we could do that is by calling the `store.dispatch()` function.

```
store.dispatch({  
  type: types.FETCH_NEW_TIME,  
  payload: new Date().toString()  
})
```

- However, this is pretty poor practice. Rather than dispatch the action directly, we'll use a function to return an action... the function will create the action (hence the name: `actionCreator`).
- This provides us with a better testing story (easy to test), reusability, documentation, and encapsulation of logic.
- Let's create our first `actionCreator` in a file called `redux/actionCreators.js`.
- We'll export a function whose entire responsibility is to return an appropriate action to dispatch on our store.

```
import * as types from './types';  
export const fetchNewTime = () => ({  
  type: types.FETCH_NEW_TIME,
```

```
payload: new Date().toString(),  
})
```

- Now if we call this function, nothing will happen except an action object is returned.
- Recall we used the `connect()` function export from `react-redux` yesterday? The first argument is called `mapStateToProps`, which maps the state to a prop object.
- The `connect()` function accepts a second argument which allows us to map functions to props as well.
- It gets called with the `dispatch` function, so here we can bind the function to call `dispatch()` on the store.
- Let's see this in action.

- In our `src/views/Home/Home.js` file, let's update our call to connect by providing a second function to use the `actionCreator` we just created. We'll call this function `mapDispatchToProps`.

```
import { fetchNewTime } from  
'../../redux/actionCreators';  
// ...  
const mapDispatchToProps = dispatch => ({  
  updateTime: () => dispatch(fetchNewTime())  
})  
// ...  
export default connect(  
  mapStateToProps,  
  mapDispatchToProps,  
) (Home);
```


- Now the `updateTime()` function will be passed in as a prop and will call `dispatch()` when we fire the action.
- Let's update our `<Home />` component so the user can press a button to update the time.

```
const Home = (props) => {  
  return (  
    <div className="home">  
      <h1>Welcome home!</h1>  
      <p>Current time: {props.currentTime}</p>  
      <button onClick={props.updateTime}>  
        Update time  
      </button>  
    </div>  
  );  
}
```

- Although this example isn't that exciting, it does showcase the features of redux pretty well.
- Imagine if the button makes a fetch to get new tweets or we have a socket driving the update to our redux store.
- This basic example demonstrates the full functionality of redux.

Multi-reducers

- As it stands now, we have a single reducer for our application.
- This works for now as we only have a small amount of simple data and (presumably) only one person working on this app.
- Just imagine the headache it would be to develop with one gigantic switch statement for every single piece of data in our apps...

- Redux to the rescue! Redux has a way for us to split up our redux reducers into multiple reducers, each responsible for only a leaf of the state tree.
- We can use the `combineReducers()` export from `redux` to compose an object of reducer functions.
- For every action that gets triggered, each of these functions will be called with the corresponding action. Let's see this in action.
- Let's say that we (perhaps more realistically) want to keep track of the current user.
- Let's create a `currentUser` redux module in... you guessed it: `src/redux/currentUser.js`:

`touch src/redux/currentUser.js`

- We'll export the same four values we exported from the `currentTime` module... of course, this time it is specific to the `currentUser`.

- We've added a basic structure here for handling a current user:

```
import * as types from './types'  
export const initialState = {  
  user: {},  
  loggedIn: false  
}  
export const reducer = (state = initialState, action) => {  
  switch (action.type) {  
    case types.LOGIN:  
      return {  
        ...state, user: action.payload, loggedIn: true};  
    case types.LOGOUT:  
      return {
```

```
...state, user: {}, loggedIn: false};  
  default:  
    return state;  
  }  
}
```

- Let's update our configureStore() function to take these branches into account, using the combineReducers to separate out the two branches

```
import { createStore, combineReducers } from 'redux';  
import { rootReducer, initialState } from './reducers'  
import { reducer, initialState as userInitialState } from  
'./currentUser'  
export const configureStore = () => {  
  const store = createStore(  

```

```
combineReducers({  
  time: rootReducer,  
  user: reducer  
}), // root reducer  
{  
  time: initialState,  
  user: userInitialState  
}, // our initialState  
);  
return store;  
}  
export default configureStore;
```

- Let's also update our Home component `mapStateToProps` function to read it's value from the time reducer

```
// ...
```

```
const mapStateToProps = state => {  
  // our redux store has `time` and `user` states  
  return {  
    currentTime: state.time.currentTime  
  };  
};
```

```
// ...
```

- Now we can create the `login()` and `logout()` action creators to send along the action on our store.

```
export const login = (user) => ({
```

```
type: types.LOGIN,  
  payload: user  
})  
// ...  
export const logout = () => ({  
  type: types.LOGOUT,  
})
```

- Now we can use the **actionCreators** to call login and logout just like the **updateTime()** action creator.
- This was another hefty day of Redux code.
- Today, we completed the circle between data updating and storing data in the global Redux state.
- In addition, we learned how to extend Redux to use multiple reducers and actions as well as multiple connected components.

- However, we have yet to make an asynchronous call for off-site data.
- Tomorrow we'll get into how to use middleware with Redux, which will give us the ability to handle fetching remote data from within our app and still use the power of Redux to keep our data.

Redux middleware

- Middleware generally refers to software services that "glue together" separate features in existing software.
- For Redux, middleware provides a third-party extension point between dispatching an action and handing the action off to the reducer:

[Action] <-> [Middleware] <-> [Dispatcher]

- Examples of middleware include logging, crash reporting, routing, handling asynchronous requests, etc.
- Let's take the case of handling asynchronous requests, like an HTTP call to a server.
- Middleware is a great spot to do this.

Our API middleware

- We'll implement some middleware that will handle making asynchronous requests on our behalf.
- Middleware sits between the action and the reducer.
- It can listen for all dispatches and execute code with the details of the actions and the current states.
- Middleware provides a powerful abstraction.

- Let's see exactly how we can use it to manage our own.
- Continuing with our `currentTime` redux work from yesterday, let's build our middleware to fetch the current time from the server we used a few days ago to actually GET the time from the API service.
- Before we get too much further, let's pull out the `currentTime` work from the `rootReducer` in the `reducers.js` file out to its own file.
- We left the root reducer in a state where we kept the `currentTime` work in the root reducer.
- More conventionally, we'll move these in their own files and use the `rootReducer.js` file (which we called `reducers.js`) to hold just the main combination reducer.

- First, let's pull the work into it's own file in `redux/currentTime.js`.
- We'll export two objects from here (and each reducer):

initialState - the initial state for this branch of the state tree

reducer - this branch's reducer

import * as types from './types';

```
export const initialState = {  
  currentTime: new Date().toString(),  
}
```

```
export const reducer = (state = initialState, action) => {  
  switch(action.type) {  
    case types.FETCH_NEW_TIME:
```

```
return { ...state, currentTime: action.payload}  
  default:  
    return state;  
}  
}
```

export default reducer

- With our currentTime out of the root reducer, we'll need to update the reducers.js file to accept the new file into the root reducer. Luckily, this is pretty easy:

```
import { combineReducers } from 'redux';  
import * as currentUser from './currentUser';  
import * as currentTime from './currentTime';  
export const rootReducer = combineReducers({
```

```
currentTime: currentTime.reducer,  
  currentUser: currentUser.reducer,  
}))  
  
export const initialState = {  
  currentTime: currentTime.initialState,  
  currentUser: currentUser.initialState,  
}  
  
export default rootReducer
```

- Lastly, let's update the configureStore function to pull the rootReducer and initial state from the file:

```
import { rootReducer, initialState } from './reducers'  
// ...  
export const configureStore = () => {  
  const store = createStore(  
    rootReducer,  
    initialState,  
  );  
  
  return store;  
}
```

Summary:



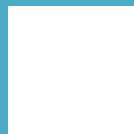
Flux



Redux



Redux Middleware



Redux actions

Thank You.....

If you have any queries please write to info@uplatz.com".