Explicació del disseny intern de l'*Issue Tracker*

Projecte laboratori - Tercer lliurament

Aplicacions i Serveis Web - Curs 2016/17 Q2

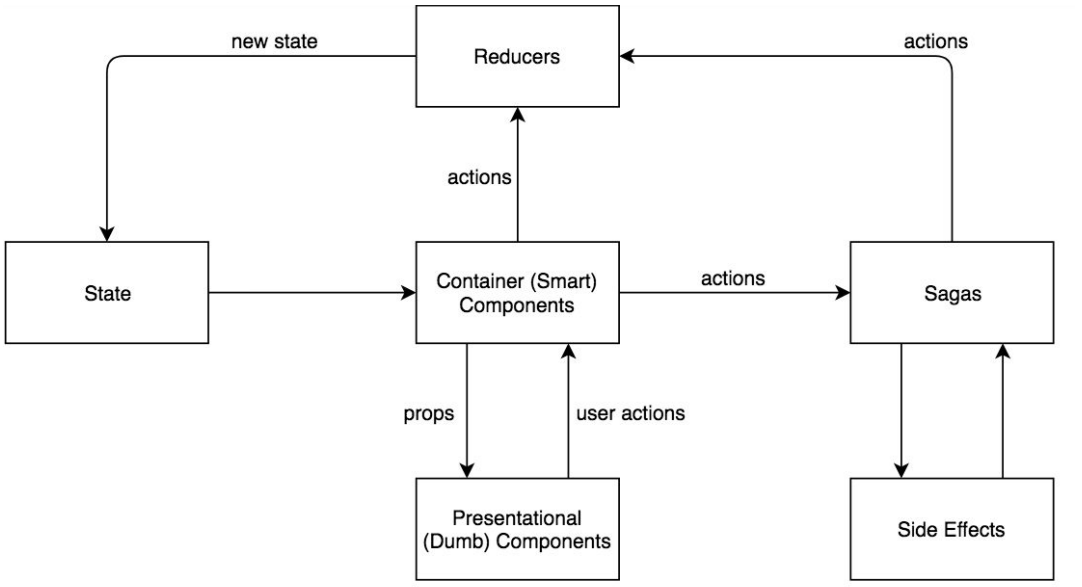
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

GRUP 12B
Iván de Mingo Guerrero
Arnau Blanch Cortès

Introducció

En aquesta entrega havíem d'implementar un client web que utilitzés l'API per a un *issue tracker* que vam realitzar en les últimes dues entregues.

Per a implementar-lo, hem escollit la llibreria *React (https://facebook.github.io/react/)* per a realitzar la vista del client. Per tal de gestionar l'estat de l'aplicació, hem utilitzat la llibreria *Redux (http://redux.js.org/)* (inspirada en l'arquitectura Flux, proposada per Facebook) en combinació amb *redux-saga (https://redux-saga.js.org/)* que permet realitzar peticions asíncrones de forma més fàcil.



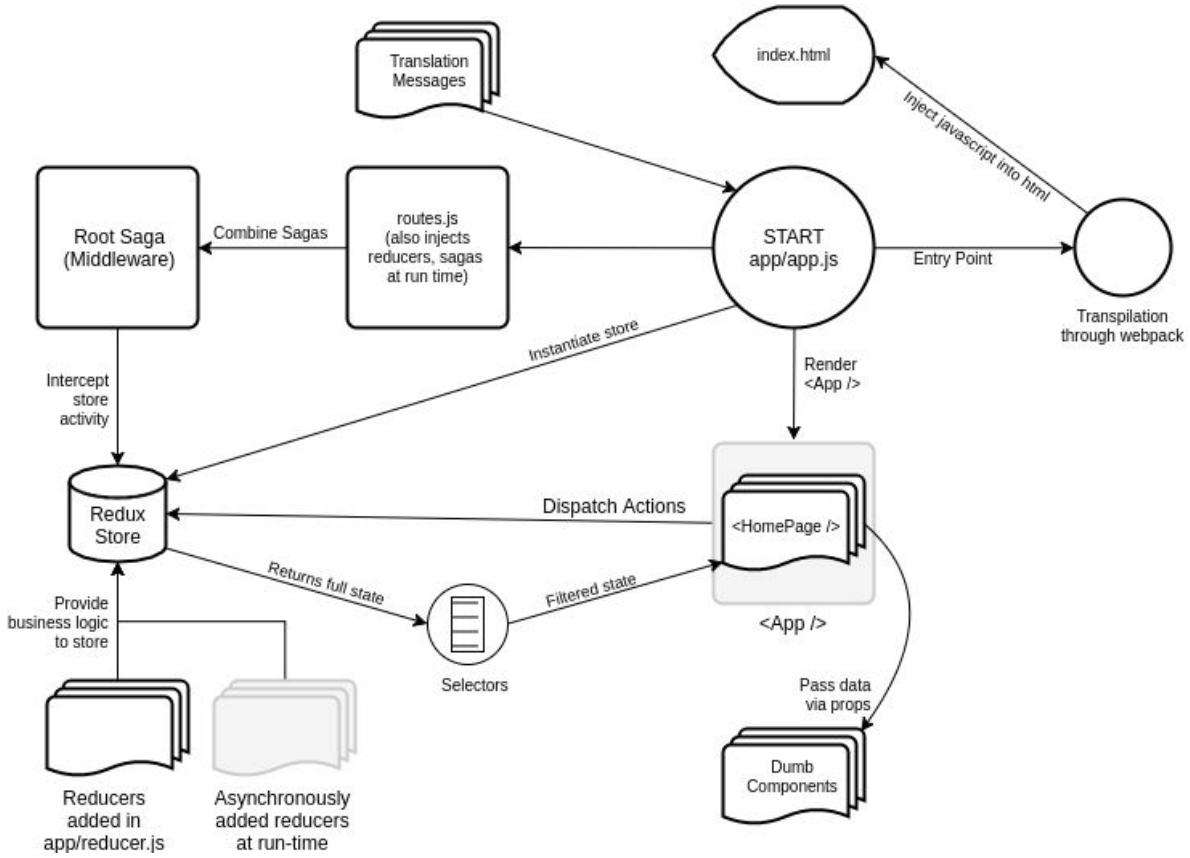
Per resumir una mica l'arquitectura, hem fet un petit exemple. La secció de comentaris de la pàgina de detalls d'una issue està representada en el disseny intern per un container. Aquest container conté diversos components que representen els comentaris i els formularis de creació i edició.

Quan un usuari ha omplert el formulari de creació, clica el botó de per enviar el comentari, aquest component passa el text del comentari a una funció que li ha passat el container de la secció de comentaris. Aleshores, el container llança una acció *CREATE_COMMENT_REQUEST* que és rebuda pel corresponent *saga* que procesa l'acció i fa la petició a l'API. Quan retorna la reposta, si la resposta és d'error, llançarà una acció *CREATE_COMMENT_FAILED* al *reducer* per modificar l'estat i indicar que hi ha hagut un error o si s'ha creat el comentari, es llançarà una acció *CREATE_COMMENT_SUCCESS* (que serà recollida pel *reducer*, que netejarà l'error) i una altra acció *GET_ISSUE_REQUEST* que serà recollida pel saga corresponent per tal que actualitzi les dades de la issue.

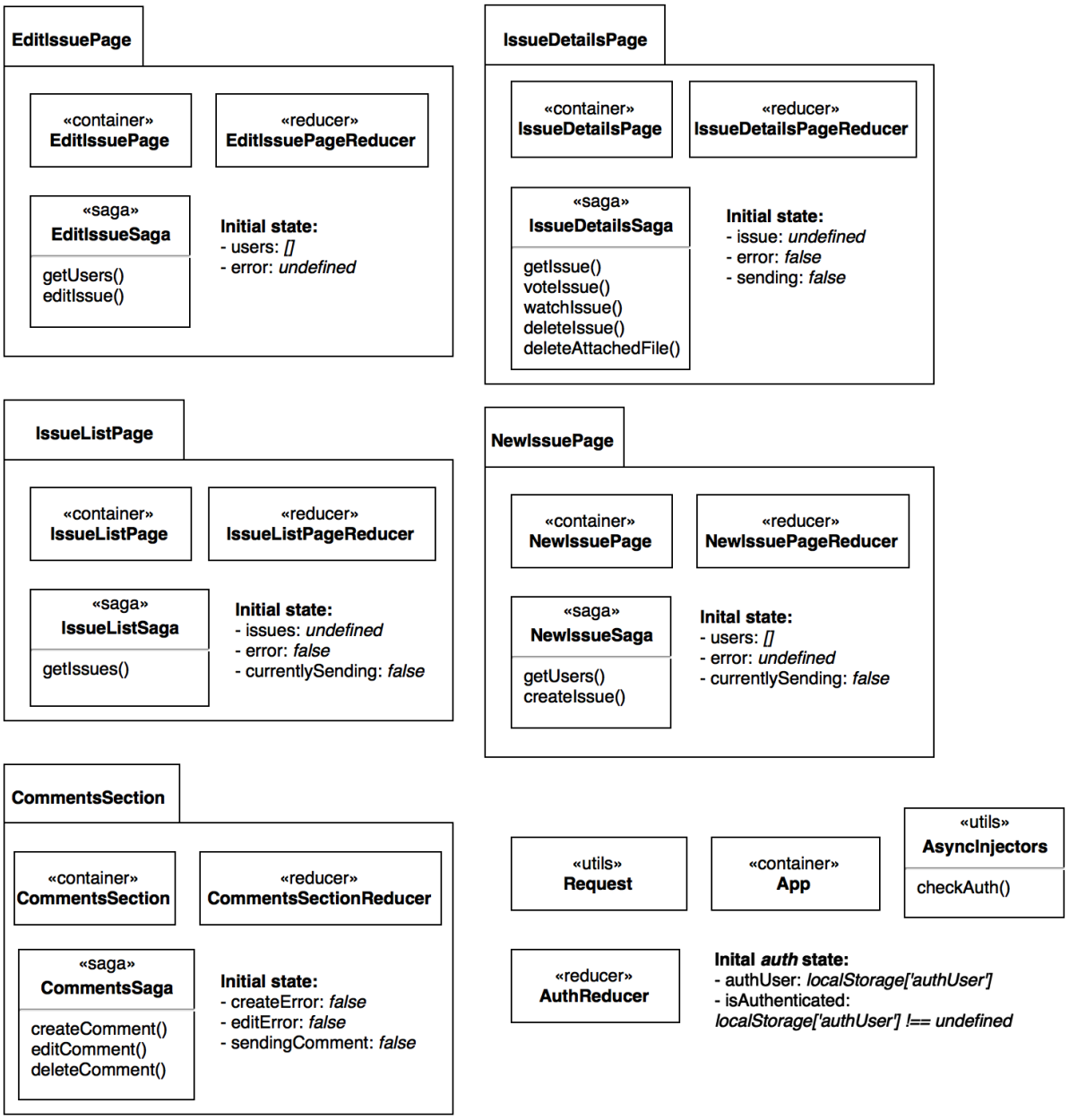
Hem partit de *react-boilerplate (https://github.com/react-boilerplate/react-boilerplate)*, que ens ha facilitat el desenvolupament al proporcionar-nos una estructura i components bàsics de

l'aplicació i algunes eines com ara un generador d'scaffolds (que genera una ruta o l'estructura de fitxers d'un component/container).

En aquest gràfic veiem de forma visual, el funcionament intern de l'aplicació. Ens podem fixar en que depenent de la ruta a la qual s'estigui accedint *s'injectaran* els reducers i sagas necessaris asíncronament.



Components interns del client web



Com que l'arquitectura que hem utilitzat en aquest client web no és l'arquitectura convencional MVC i que els components no es comuniquen entre ells directament, sinó que passen per intermedaris no hem fet un diagrama de classes com els que hem estudiat.

Per aquesta raó, hem decidit mostrar les diferents parts de l'aplicació i els seus components interns (només hem representat les parts que hem implementat nosaltres, exceptuant els components i algun altre fitxer).

El container *App* és el que està renderitzat tota l'estona ja que conté l'*AppHeader* (i s'encarrega de llançar les peticions de canvi d'autenticació) i els seus *fills* són la resta de containers, depenent de la ruta actual.

La visualització del llistat d'issues està englobat al container *IssueListPage*, aquest container té els seus *reducer* i *saga* que s'encarreguen de realitzar els canvis a l'*state* d'aquesta part de l'aplicació i a fer les peticions a l'API que requereix, respectivament.

El mateix passa amb la visualització detallada d'una issue i l'esborrat d'adjunts que està contingut a *IssueDetailsPage*, amb la creació d'issues a *NewIssuePage*, amb l'edició d'issues a *EditIssuePage* i la secció de comentaris d'una issue (visualització, creació, edició i esborrat) a *CommentsSection*.

Cada saga d'un container es divideix la seva execució (*fork*) en els seus sagas interns, que es quedaran parats esperant que arribi l'acció que els activa, realitzaran la seva feina i tornaran a parar-se per esperar l'acció.

L'*state* de l'aplicació està dividit en diversos *substates*: un per a cada part del client web, un per a l'autenticació i alguns altres (per al router, la gestió dels formularis amb *redux-form*, etc).

Cada *substate* serà modificat pel seu reducer corresponent quan rebi una de les accions que espera. A més, aquest *substate* és accedit pels containers, a través d'un selector (que filtra l'*state* de tota l'aplicació), per passar-ne una part als seus components) . Quan hi ha un canvi a l'*state*, les vistes es re-renderitzen automàticament per reflectir els nous canvis.

Alguns d'aquests processos es poden veure al segon esquema de la introducció.

Per tal de facilitar les crides a l'API, hem creat la funció *request* que agafa del localStorage l'usuari que està seleccionat (si se n'ha seleccionat un) i n'agafa l'API key de la llista d'usuaris *hardcodejats*. Aleshores, afegeix aquest header i algun extra (*Accept* i/o *Content-Type*), realitzar la petició utilitzant *Fetch* i en retorna la *promise*.

Per a l'autenticació, hem *hardcodejat* uns usuaris que es poden seleccionar des de la barra superior. Afegint una funció nova a *asyncInjectors* hem pogut modificar algunes rutes de l'aplicació per tal que requereixin haver seleccionat un usuari.

Processament de peticions a l'API

Nota: En els següents diagrames, hem obviat els intermediaris de Redux entre containers, sagas i reducers i enlloc de fer 'dispatch' d'una acció, es fa una crida directa per tal de simplificar el diagrama i fent-lo més entenedor.

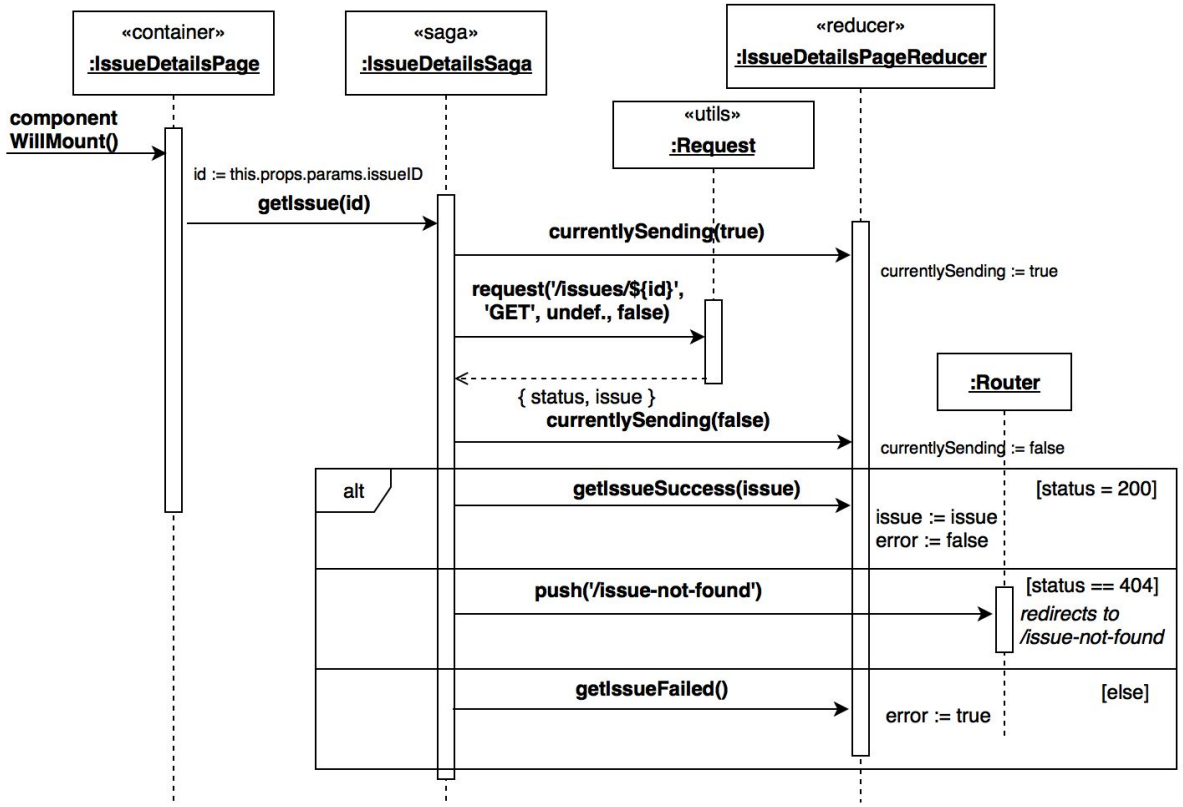
Petició GET /issues/:id

Quan l'usuari vol veure la pàgina de detalls d'una issue, es carrega el *container IssueDetailsPage*. Quan es fa el montatge del *container*, s'invoca la funció *componentWillMount()*, la qual llança l'acció *GET_ISSUE_REQUEST* que és recollida pel *saga IssueDetailsSaga* per tal d'obtenir la informació de la *issue*.

A countinuació, el *saga* avisa el *reducer* que està carregant la *issue* (*currentlySending*) per tal que modifiqui l'*state* i es mostri un indicador de *loading* a la pàgina. Llavors s'utilitza la funció *request* amb els paràmetres pertinents per tal de realitzar la crida a l'API. Quan es rep la resposta ja es pot processar el resultat.

Depenent del codi d'estat de la resposta, es dispara una acció o una altra. En aquest cas, si la resposta ha estat satisfactòria (codi 200) es dispara l'acció *GET_ISSUE_SUCCESS*, que envia la informació de la *issue* al *reducer*. Aquest, modificarà l'*state* i es veurà reflectit en la vista.

Si la resposta té el codi 404 *Not Found*, llavors es redirecciona a una pàgina que mostri que no s'ha trobat l'issue (i per tant, la pàgina no existeix). Per qualsevol altra resposta, es dispara l'acció *GET_ISSUE_FAILED*.



Petició POST /issues

La petició per crear una nova *issue* es crida quan l'usuari, després d'omplir el formulari correctament, prem el botó "*CREATE ISSUE*". Concretament, s'invoca la funció *onSubmit()* del *component IssueForm* (el formulari per crear una *issue*), la qual llança l'acció *CREATE_ISSUE_REQUEST* que és rebuda pel saga *NewIssueSaga* passant-li un objecte amb la informació de la issue que l'usuari ha introduït al formulari.

A continuació, el *saga* dispara l'acció *CURRENTLY_SENDING* per tal que es mostri un *loading* a la pàgina fins que no es rebi la resposta de l'API. Seguidament, es realitza la petició.

Quan es rep la resposta de l'API, el *saga* dispara diferents accions segons el codi d'estat d'aquesta, a més de *CURRENTLY_SENDING* amb *false*. Per una banda, si el codi és 201 *Created*, es dispara l'acció *CREATE_ISSUE_SUCCESS* i es redirigeix a la pàgina */issues/${issue.id}* per veure la informació rebuda de la *issue*. En canvi, per qualsevol altre codi d'status, es dispara l'acció *CREATE_ISSUE_FAILURE*.