Esercizio 4 – RIA

Matteo Bettini – Andrea Aspesi

Esercizio 4: trasferimento denaro (Pure HTML)

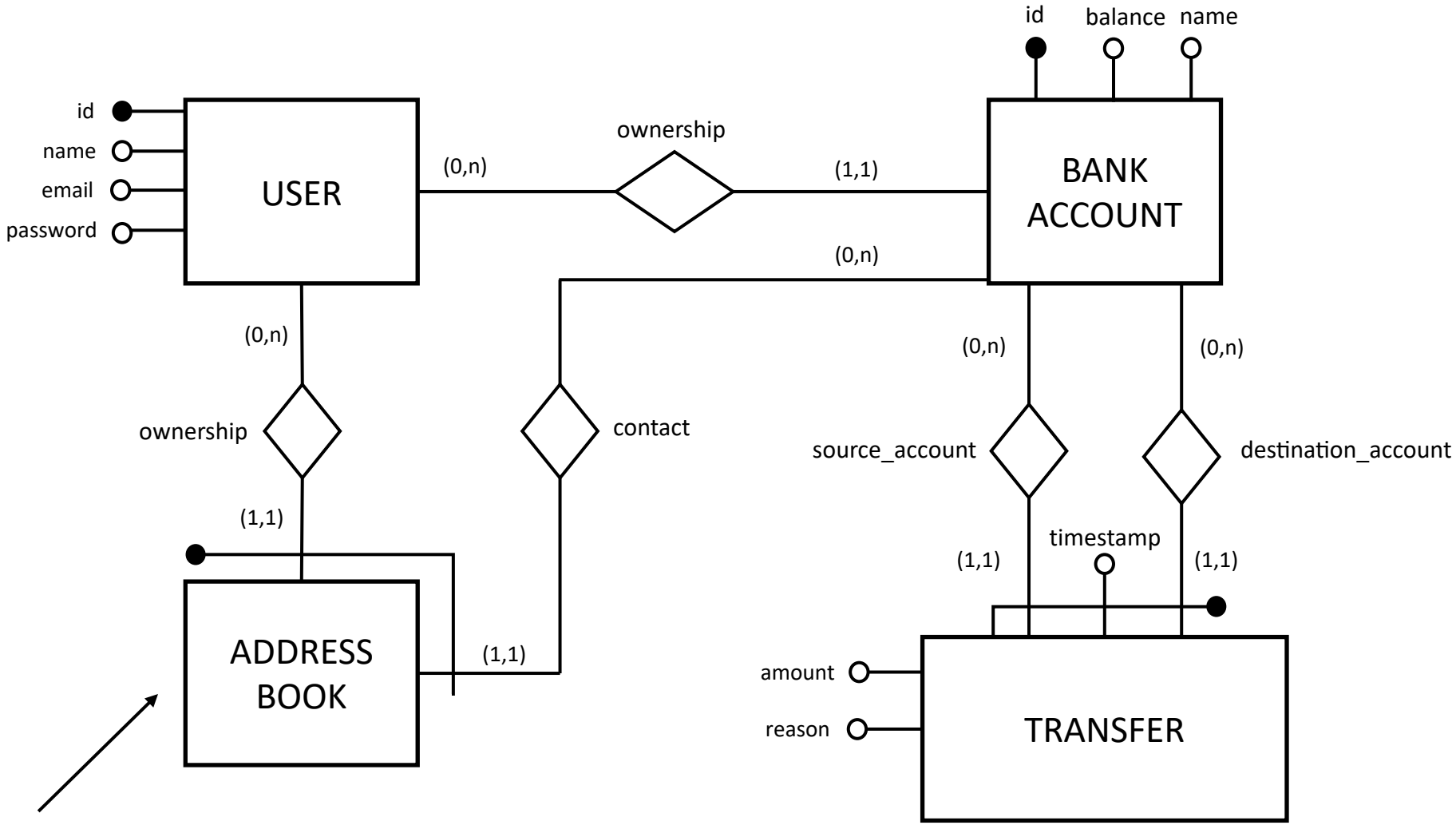
- Un'applicazione web consente la gestione di trasferimenti di denaro online da un conto a un altro. Un utente ha un nome, un codice e uno o più conti correnti. Un conto ha un codice, un saldo, e i trasferimenti fatti (in uscita) e ricevuti (in ingresso) dal conto. Un trasferimento ha una data, un importo, un conto di origine e un conto di destinazione. Quando l'utente accede all'applicazione appare una pagina LOGIN per la verifica delle credenziali. In seguito all'autenticazione dell'utente appare l'HOME page che mostra l'elenco dei suoi conti. Quando l'utente seleziona un conto, appare una pagina STATO DEL CONTO che mostra i dettagli del conto e la lista dei movimenti in entrata e in uscita, ordinati per data discendente. La pagina contiene anche una form per ordinare un trasferimento. La form contiene i campi: codice utente destinatario, codice conto destinatario, causale e importo. All'invio della form con il bottone INVIA, l'applicazione controlla che il conto di destinazione appartenga all'utente specificato e che il conto origine abbia un saldo superiore o uguale all'importo del trasferimento. In caso di mancanza di anche solo una condizione, l'applicazione mostra una pagina con un avviso di fallimento che spiega il motivo del mancato trasferimento. In caso di verifica di entrambe le condizioni, l'applicazione deduce l'importo dal conto origine, aggiunge l'importo al conto destinazione e mostra una pagina CONFERMA TRASFERIMENTO che presenta i dati del conto di origine e destinazione, con i rispettivi saldi aggiornati. L'applicazione deve garantire l'atomicità del trasferimento: ogni volta che il conto di destinazione viene addebitato il conto di origine deve essere accreditato e viceversa.
- Entities, attributes, relationships

Esercizio 4: trasferimento denaro (RIA)

- L'applicazione supporta registrazione e login mediante una pagina pubblica con opportune form. La registrazione controlla la validità sintattica dell'indirizzo di email e l'uguaglianza tra i campi "password" e "ripeti password", anche a lato client. La registrazione controlla l'unicità dello username.
- Dopo il login, l'intera applicazione è realizzata con un'unica pagina.
- Ogni interazione dell'utente è gestita senza ricaricare completamente la pagina, ma produce l'invocazione asincrona del server e l'eventuale modifica del contenuto da aggiornare a seguito dell'evento.
- I controlli di validità dei dati di input (ad esempio importo non nullo e maggiore di zero) devono essere realizzati anche a lato client.
- L'avviso di fallimento è realizzato mediante un messaggio nella pagina che ospita l'applicazione.
- L'applicazione chiede all'utente se vuole inserire nella propria rubrica i dati del destinatario di un trasferimento andato a buon fine non ancora presente. Se l'utente conferma, i dati sono memorizzati nella base di dati e usati per semplificare l'inserimento. Quando l'utente crea un trasferimento, l'applicazione propone mediante una funzione di auto-completamento i destinatari in rubrica il cui codice corrisponde alle lettere inserite nel campo codice destinatario.

- Entities, attributes, relationships

Database Design



Many-many relation,
resolved as an entity in the schema

Database Schema (1/3)

```
create table `user` (
  `id` int not null auto_increment,
  `name` varchar(40) not null,
  `email` varchar(40) not null,
  `password` varchar(40) not null,
  primary key (`id`),
  unique(`email`)
) engine=InnoDB;
```

```
create table `bank_account` (
  `id` int not null auto_increment,
  `userid` int not null,
  `name` varchar(40) not null,
  `balance` decimal(10,2) not null default '0.00',
  primary key (`id`),
  constraint `accountOfUser` foreign key (`userid`)
  references `user`(`id`) on delete cascade
                          on update cascade,
  constraint `positiveOrZeroBalance` check (`balance` >= 0)
) engine=InnoDB;
```

Database Schema (2/3)

```
create table `transfer` (
  `source_account` int not null,
  `destination_account` int not null,
  `timestamp` timestamp not null default current_timestamp,
  `reason` varchar(255) not null,
  `amount` decimal(10,2) not null,
  primary key (`source_account`,`destination_account`,`timestamp`),
  constraint `destinationAccount` foreign key (`destination_account`)
  references `bank_account` (`id`) on delete cascade
                                   on update cascade,
  constraint `sourceAccount` foreign key (`source_account`)
  references `bank_account` (`id`) on delete cascade
                                   on update cascade,
  constraint `positiveAmount` check (`amount` > 0)
) engine=InnoDB;
```

Database Schema (3/3)

```
create table `address_book` (
  `owner_id` int not null,
  `contact_account` int not null,
  primary key (`owner_id`,`contact_account`),
  constraint `ownerUser` foreign key (`owner_id`)
  references `user` (`id`) on delete cascade
                           on update cascade,
  constraint `contactAccount` foreign key (`contact_account`)
  references `bank_account` (`id`) on delete cascade
                                   on update cascade
) engine=InnoDB;
```

Analisi Requisiti Applicazione

- L'applicazione supporta registrazione e login mediante una pagina pubblica con opportune form. La registrazione controlla la validità sintattica dell'indirizzo di email e l'uguaglianza tra i campi "password" e "ripeti password", anche a lato client. La registrazione controlla l'unicità dello username.
- Dopo il login, l'intera applicazione è realizzata con un'unica pagina.
- Ogni interazione dell'utente è gestita senza ricaricare completamente la pagina, ma produce l'invocazione asincrona del server e l'eventuale modifica del contenuto da aggiornare a seguito dell'evento.
- I controlli di validità dei dati di input (ad esempio importo non nullo e maggiore di zero) devono essere realizzati anche a lato client.
- L'avviso di fallimento è realizzato mediante un messaggio nella pagina che ospita l'applicazione.
- L'applicazione chiede all'utente se vuole inserire nella propria rubrica i dati del destinatario di un trasferimento andato a buon fine non ancora presente. Se l'utente conferma, i dati sono memorizzati nella base di dati e usati per semplificare l'inserimento. Quando l'utente crea un trasferimento, l'applicazione propone mediante una funzione di auto-completamento i destinatari in rubrica il cui codice corrisponde alle lettere inserite nel campo codice destinatario.

- pages (views), view components, events, actions

Completamento delle Specifiche

- Ogni utente viene identificato dal suo codice utente o dalla sua email.
- Per credenziali di login si intendono email e password. Per username si intende l'email (su cui viene fatto il controllo di unicità).
- Se l'utente è loggato, verrà reindirizzato automaticamente alla pagina protetta se tenta di accedere alla pagina di login/registrazione. Viceversa se l'utente non è loggato, verrà reindirizzato alla pagina di login/registrazione.
- Se l'utente è loggato, è possibile fare logout e tornare quindi alla pagina di login
- Dopo una registrazione avvenuta con successo, viene fatto login automaticamente.
- Non è possibile fare trasferimenti con conto origine uguale al conto destinazione.
- L'avviso di conferma trasferimento è realizzato mediante un messaggio nella pagina che ospita l'applicazione.
- Per migliorare la presentazione ogni conto ha un nome che sarà univoco per ogni utente.
- Dalla pagina home, è anche possibile creare un nuovo conto specificando il nome; se esiste già un conto di quell'utente con lo stesso nome, ciò verrà notificato, altrimenti il conto verrà creato (e la pagina aggiornata in modo asincrono).
- Durante l'inserimento del codice del destinatario, vengono presentati come suggerimenti in un menu a tendina l'elenco dei destinatari in rubrica. Selezionato il destinatario, viene presentati come suggerimenti in un menu a tendina i suoi conti in rubrica.

Further additions

To better enrich the user's experience we've made the following additions to the specifications:

- For every input field in the application we intercept the pression of the enter key and, after preventing the default submit effect, we simulate a "click" event on the correspondent custom submit button of the form.
- To every ajax request it is associated a loading graphic in the bottom-left corner which keeps the user updated on the status of the request.
- The user has the possibility to display and hide the "create account" form and the "make transfer" form with an associated "display/hide" button.
- When the user selects an account, the selection button changes from "open" to "hide". The user is therefore able to display and hide account's details at his pleasure.
- When adding a contact to the address book, the user is graphically informed on the status and on the outcome of the operation.

Server-Side Components

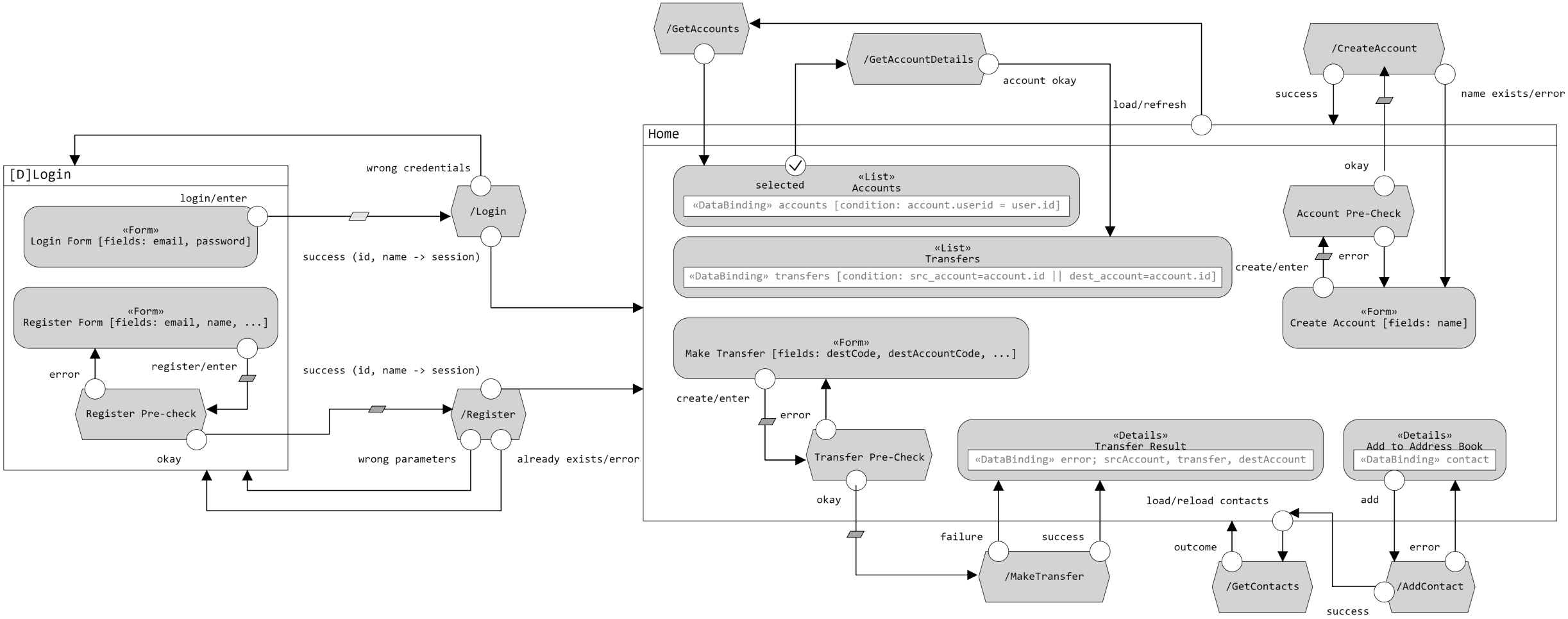
- **Model objects (Beans)**
  - User
  - BankAccount
  - Transfer
  - AddressBook
- **Data access objects (DAO)**
  - UserDAO
    - findUser(email, password) : User
    - getUserByID(id) : User
    - getUserByEmail(email) : User
    - registerUser(name, email, password)
  - BankAccountDAO
    - getAccountByID(id) : BankAccount
    - getAccountByUserIDAndName(id) : BankAccount
    - getAccountsByUserID(userId) : List<BankAccount>
    - createAccount(userId, name)
  - TransferDAO
    - getTransfersByAccountId(accountId): List<Transfer>
    - makeTransfer(sourceAccountId, destAccountId, reason, amount)
  - AddressBookDAO
    - getAddressBookByOwnerID(ownerId): AddressBook
    - addContactToAddressBook(userId, destAccountId)
    - existsContactEntry(ownerId, contact_account): boolean
- **Filters**
  - checkLoggedUser
  - checkNotLoggedUser
  - NoCacher
- **Controllers (servlets) [access rights]**
  - Login [not logged user]
  - Logout [all]
  - Register [not logged user]
  - GetAccounts [logged user]
  - GetContacts [logged user]
  - AddContact [logged user]
  - CreateAccount [logged user]
  - GetAccountDetails [logged user]
  - MakeTransfer [logged user]
- **Views (Templates)**
  - login.html
  - home.html

Client-Side Components

- **Login (Index)**
  - Login form
    - manage submit and errors
  - Register form
    - manage submit and errors

- **Home**
  - PageOrchestrator
    - **start():** initializes page components
    - **refresh(excludeContacts):** loads page components, excluding loading the address book if the flag is set
  - UserInfo
    - **show():** shows user's data in page header
  - AccountList
    - **show():** loads accounts
    - **update():** refreshes accounts' view with accounts' data
  - TransferList
    - **show(accountID):** loads transfers of the provided account id
    - **hide():** hides account details view
    - **update():** refreshes account details view with transfers' data
  - TransferResult
    - **showSuccess(srcAccount, transfer, destAccount):** shows transfer's outcome in case of success
    - **showFailure(reason):** shows transfer's failure reason
  - AddressBook
    - **load():** loads contacts from server
    - **showButton(destUserCode, destAccountCode):** shows add contact button when contact is not already in the address book
    - **addContact(destUserCode, destAccountCode):** add contact to address book
    - **autocompleteDest(destUserCode):** provides autosuggestion options for destination users for transfers
    - **autocompleteAccount(destUserCode, destAccountCode, currentAccount):** provides autosuggestion options for destination accounts for transfers (excluding current account)

Design Applicativo (IFML)



Interactions with hidden page components and server are semplified here. Refer to sequence diagrams.

Events & Actions

Client side		Server side	
Evento	**Azione**	**Evento**	**Azione**
Login.html → login form → submit	Data check	POST (username, password)	Data check
Login.html → register form → submit	password match and data check	POST (credentials)	Data check
Home.html → load	Update view -> update account list and contacts	2 GET no param	Retrieval of data from db and Json conversion
Home.html → account list → account selection	Update view and display transfers of selected account	GET (account id)	Retrieval of data from db and Json conversion
Press enter in an input filed	Click the correspondent custom submit button	-	-
Form hide/show buttons	Hide/show correspondent form	-	-
Home.html → Make transfer → submit	Data check	POST (transfer data)	Transfer check and outcome
Home.html → successResult → add contact	Ajax post	POST (contact info)	Addition of contact
Home.html → Transfer list → transfer success	Optionally show «add contact» button & show success	-	-
Home.html → Account list → create account	Local check on name	POST (account name)	Data check & insertion
Home.html → Transfer list → insert input in make transfer form	Process and display suggestions	-	-

Controller & Event Handlers

Client side		Server side	
Evento	**Controllore**	**Evento**	**Controllore**
index → login form → submit	Function makeCall	POST (username password)	Login (servlet)
index → register form → submit	Function makeCall	POST (name, email, password)	Register (servlet)
home → load	Function PageOrchestrator -> AccountList -> AddressBook	 GET (no parameters) GET (no parameters)	 GetAccounts (servlet) GetContacts (servlet)
home → accounts list → select account	Function AccountList -> TransferList	 GET (accountID)	 GetAccountDetails (servlet)
home → account form → submit	Function AccountList	POST (accountName)	CreateAccount (servlet)
home → transfer form → submit	Function TransferList	POST (transfer data)	MakeTransfer (servlet)
home → transfer success div → add contact	Function AddressBook	POST (contact data)	AddContact (servlet)
home → transfer form → click / typing on destCode	Function TransferList -> AddressBook	-	-
home → transfer form → click / typing on destAccountCode	Function TransferList -> AddressBook	-	-

Sequence diagrams

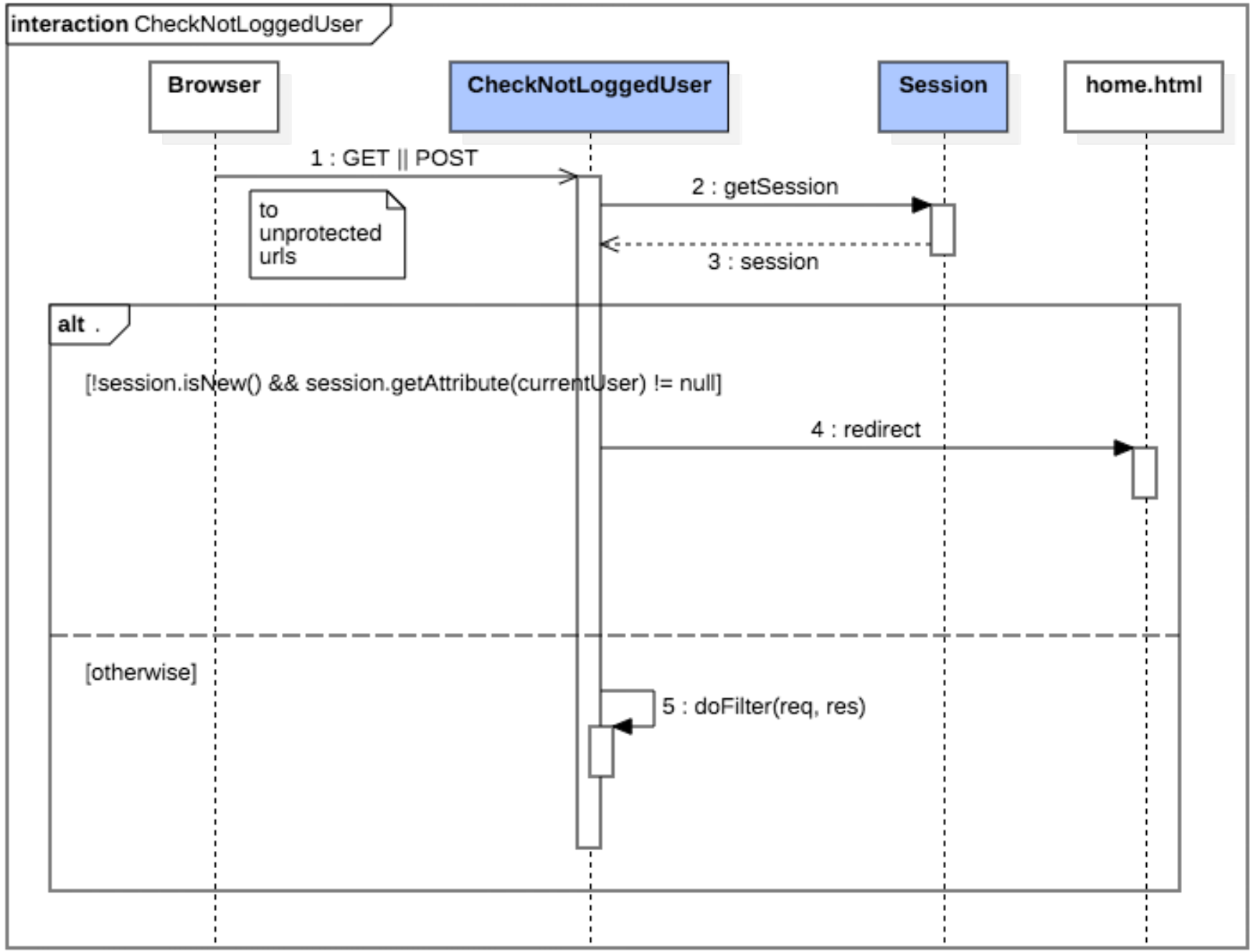
The following sequence diagrams aim to depict the interaction flow underlying the core events of the web application.
Although the authors tried their best to pursue and achieve full clarity, some minor details have been left out due to their repetitiveness (e.g. internal server errors, database access errors, null parameters).
Specifically, database access errors always lead to an answer with code 500, which is always displayed in the correspondent warning section of the page.

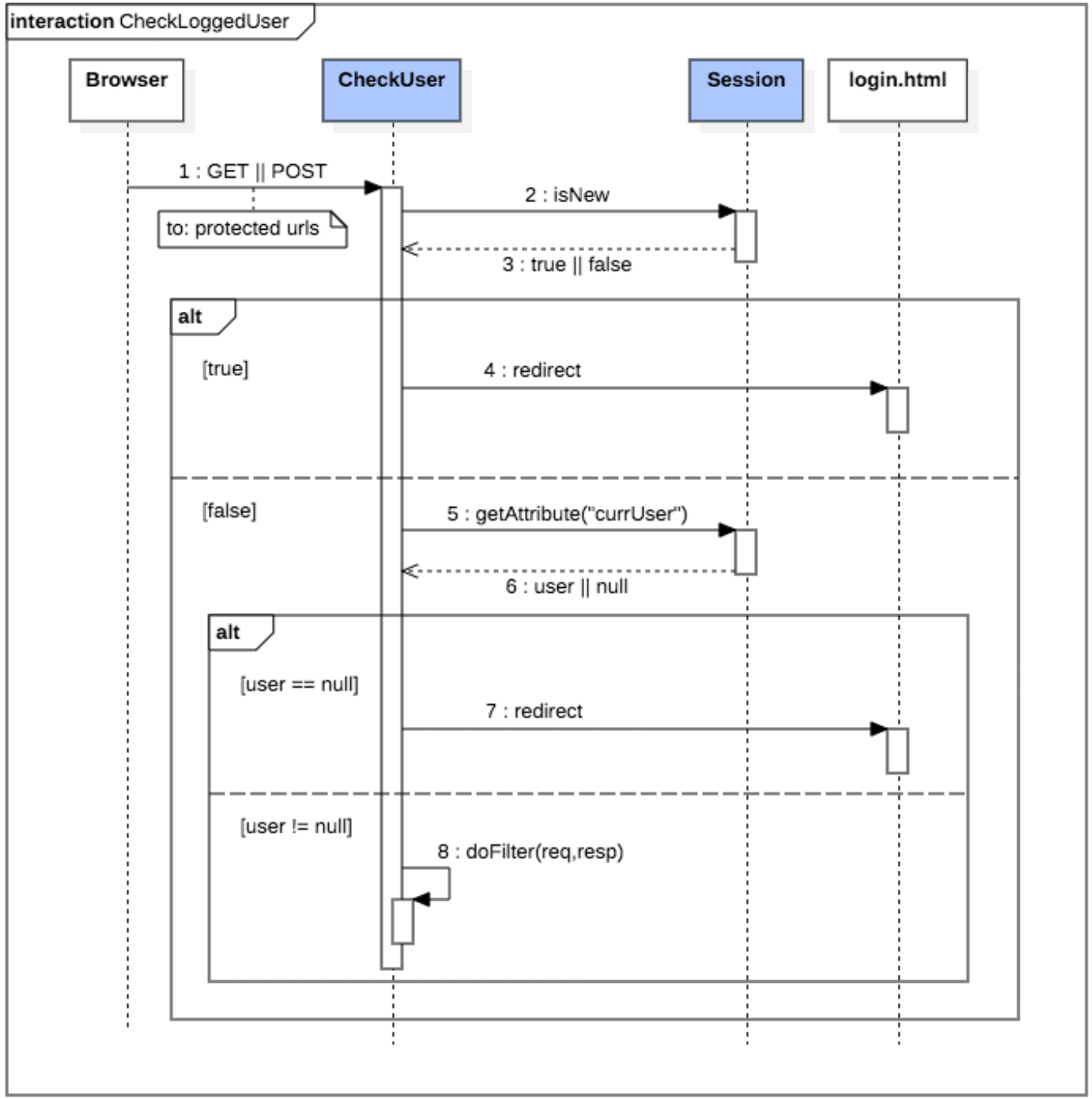
Minor graphical event handling interactions, which do not involve AJAX asynchronous calls, may have also been omitted due to their simple and secondary role.

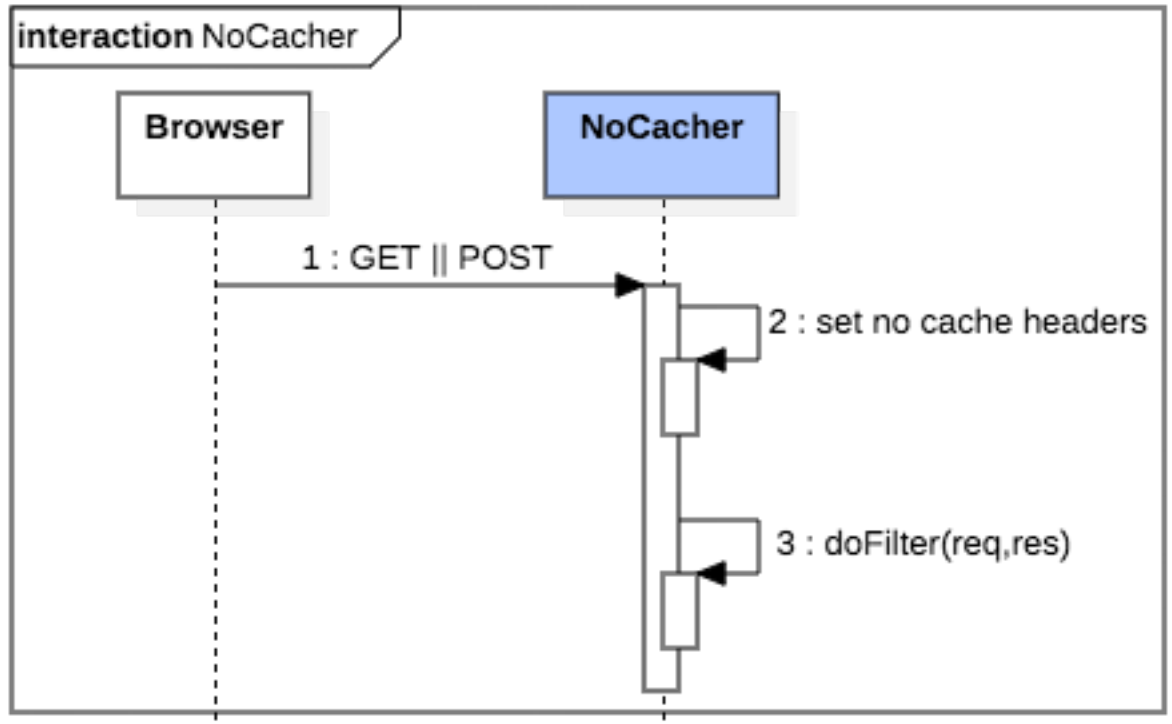
Furthermore, filter checks are represented in the first diagrams and are omitted in the subsequent ones, which does not entail that they are not called when supposed to.

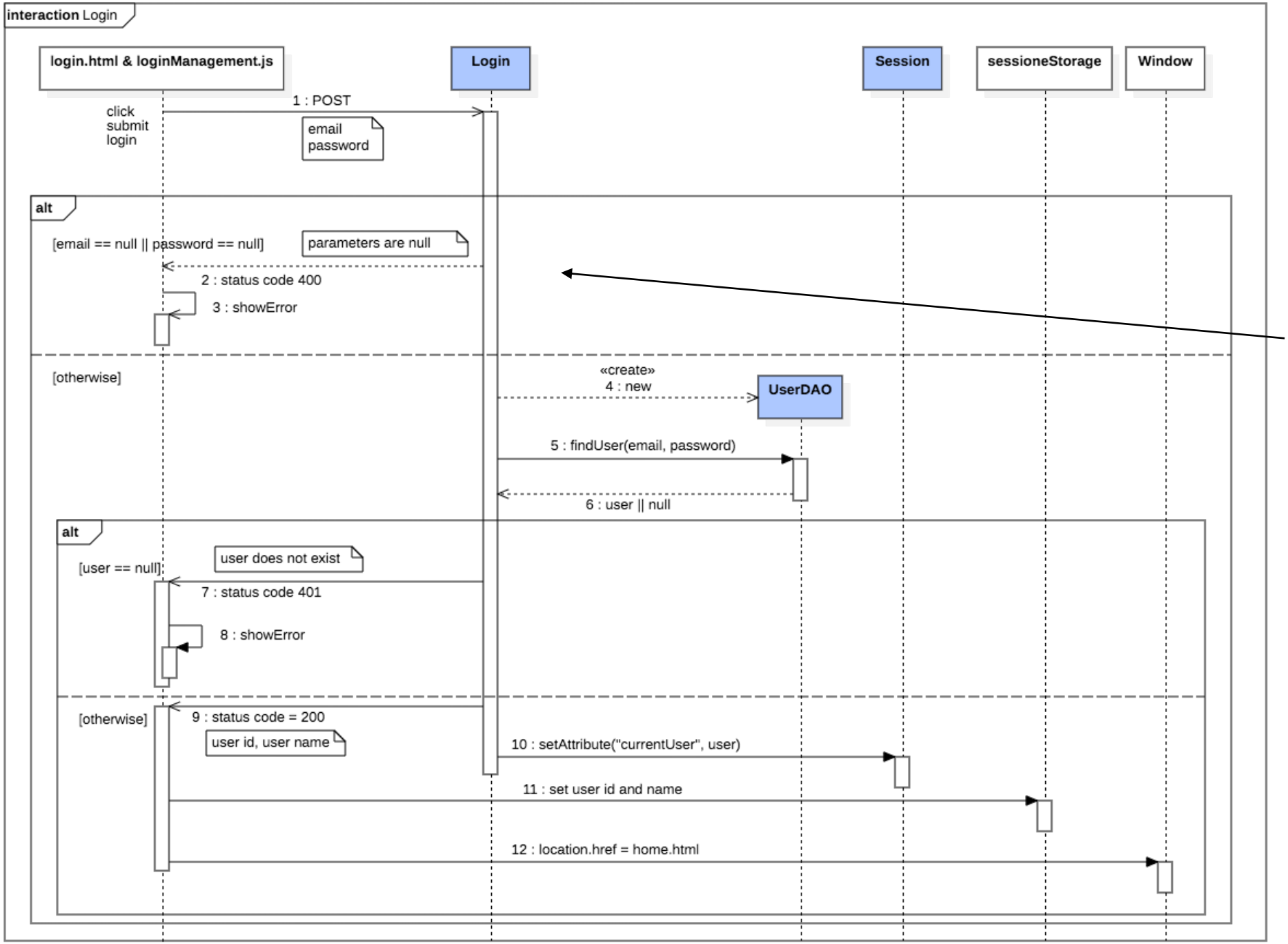
The name of the event is in the top-left corner.

Blue lifelines indicate server-side components, while white lifelines indicate client-side components.

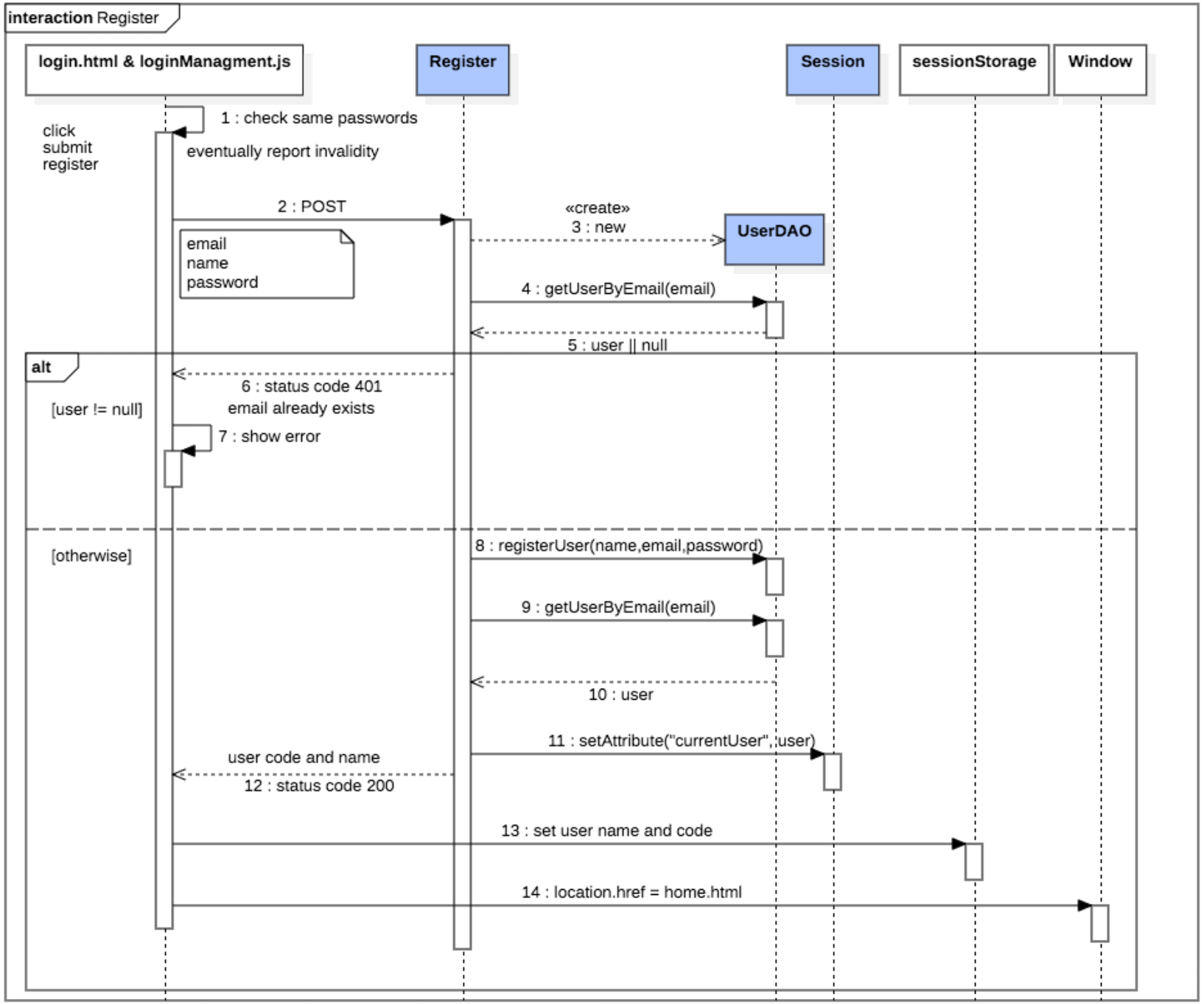

interaction CheckNotLoggedUser
Browser
CheckNotLoggedUser
Session
home.html
1 : GET || POST
to unprotected urls
2 : getSession
3 : session
alt .
[!session.isNew() && session.getAttribute(currentUser) != null]
4 : redirect
[otherwise]
5 : doFilter(req, res)

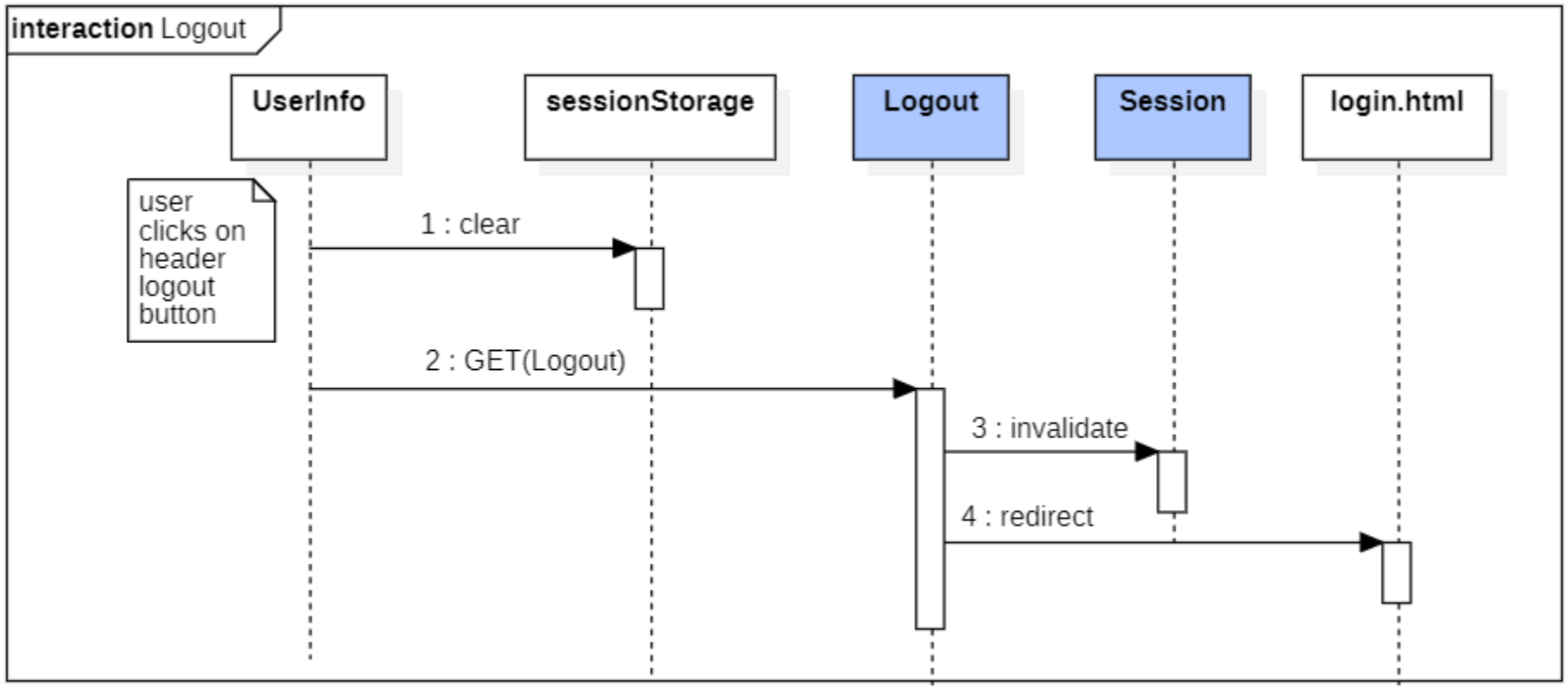

interaction CheckLoggedUser
Browser
CheckUser
Session
login.html
1 : GET || POST
to: protected urls
2 : isNew
3 : true || false
alt
[true]
4 : redirect
[false]
5 : getAttribute("currUser")
6 : user || null
alt
[user == null]
7 : redirect
[user != null]
8 : doFilter(req,resp)

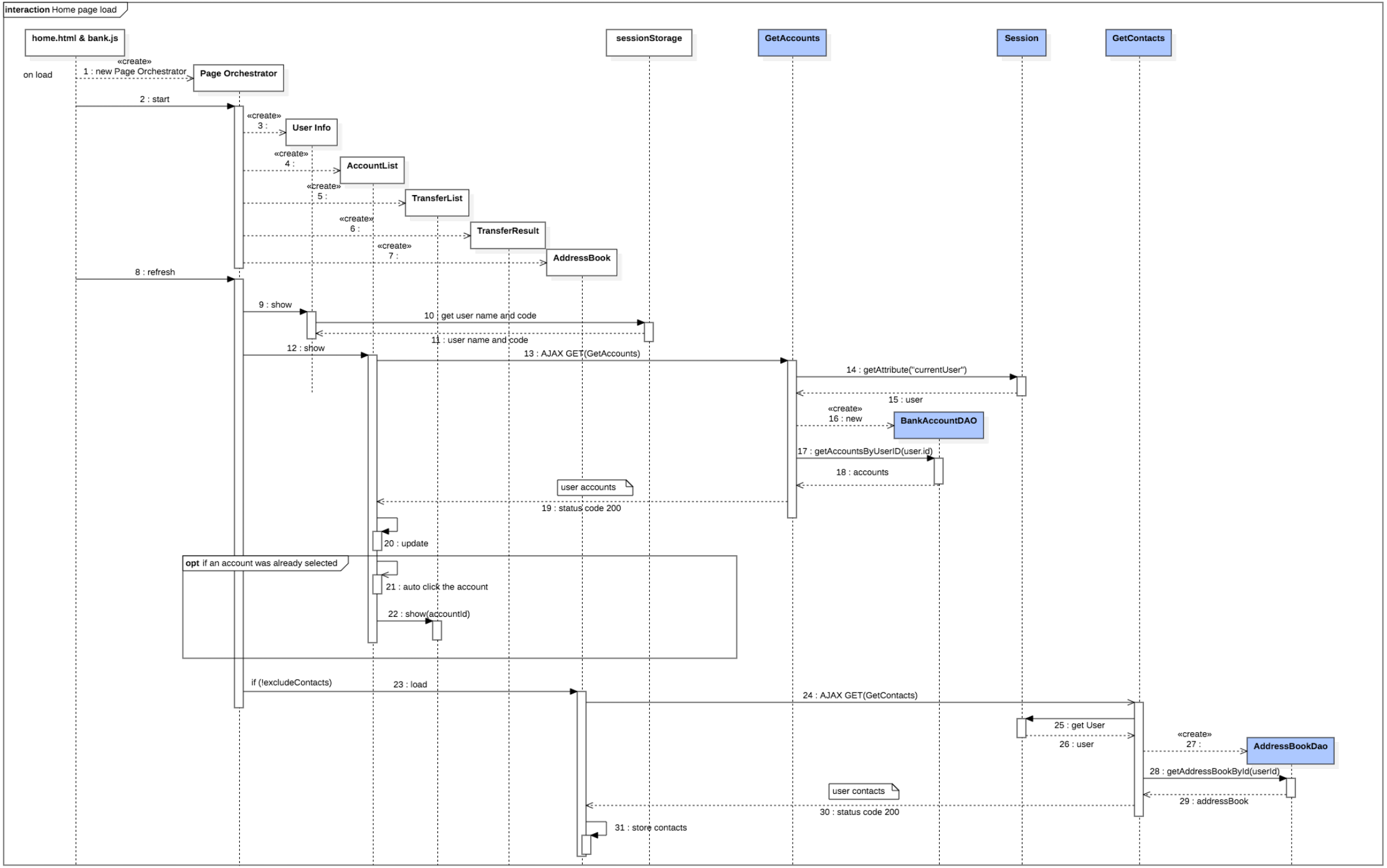

interaction NoCacher
Browser
NoCacher
1 : GET || POST
2 : set no cache headers
3 : doFilter(req,res)

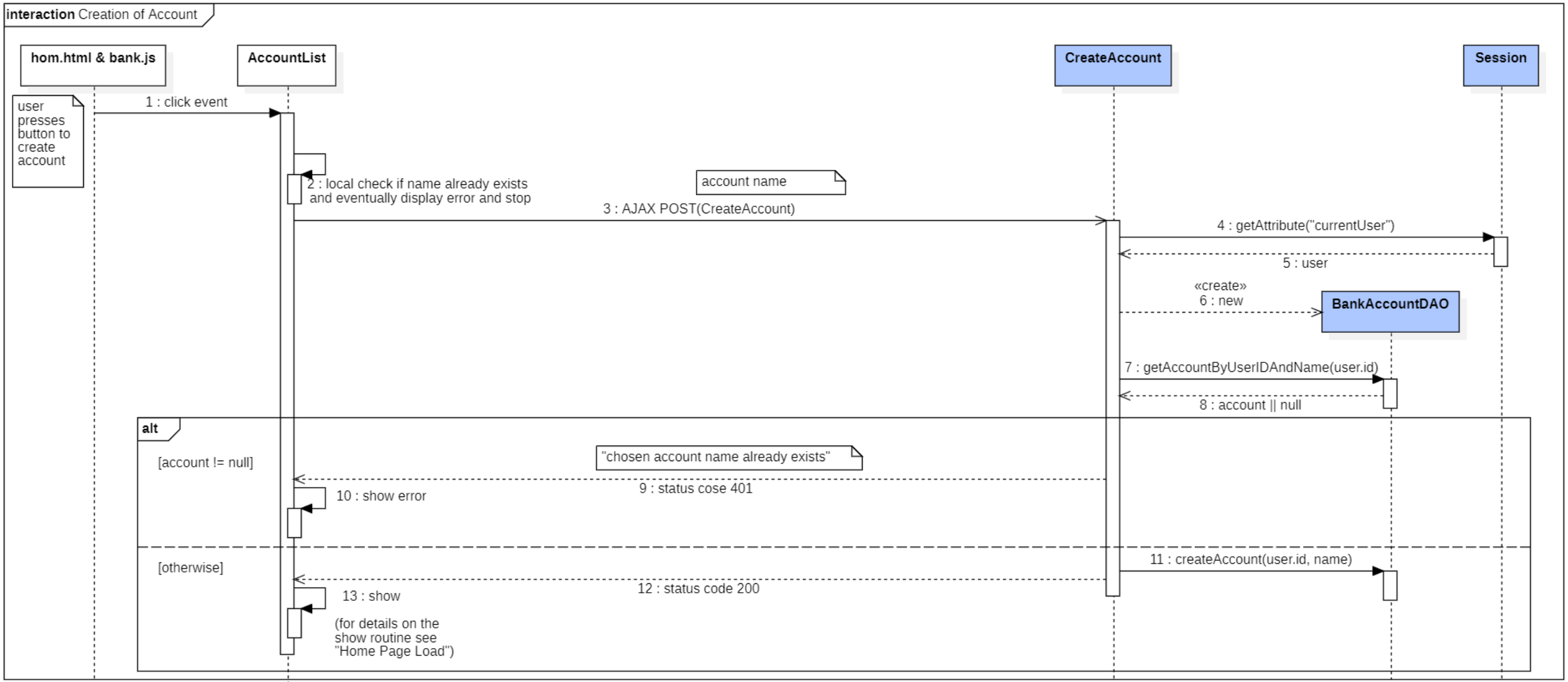


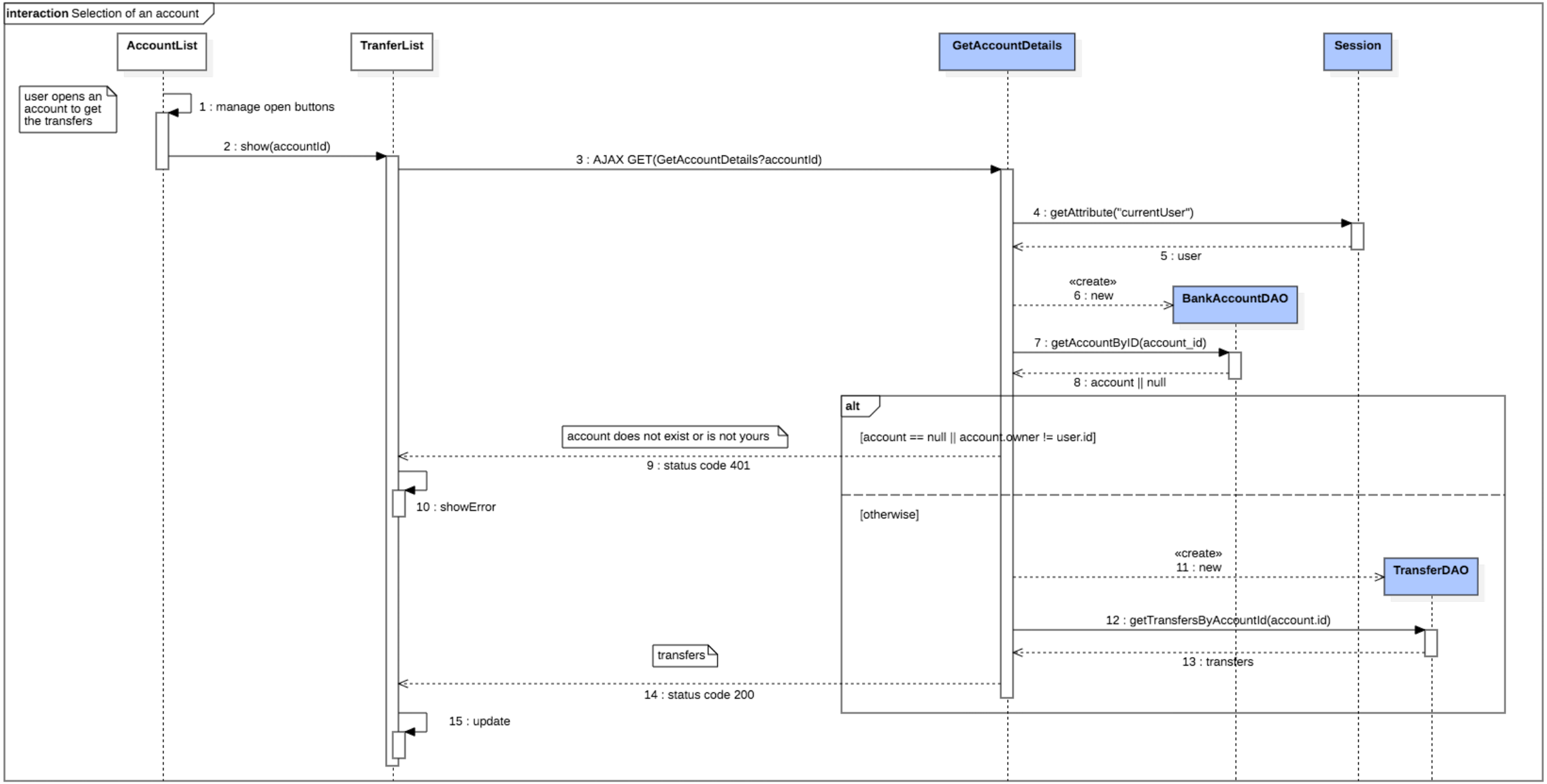
Invalid or null parameters checks will be omitted in the following sequence diagrams, as they are always treated like in this case

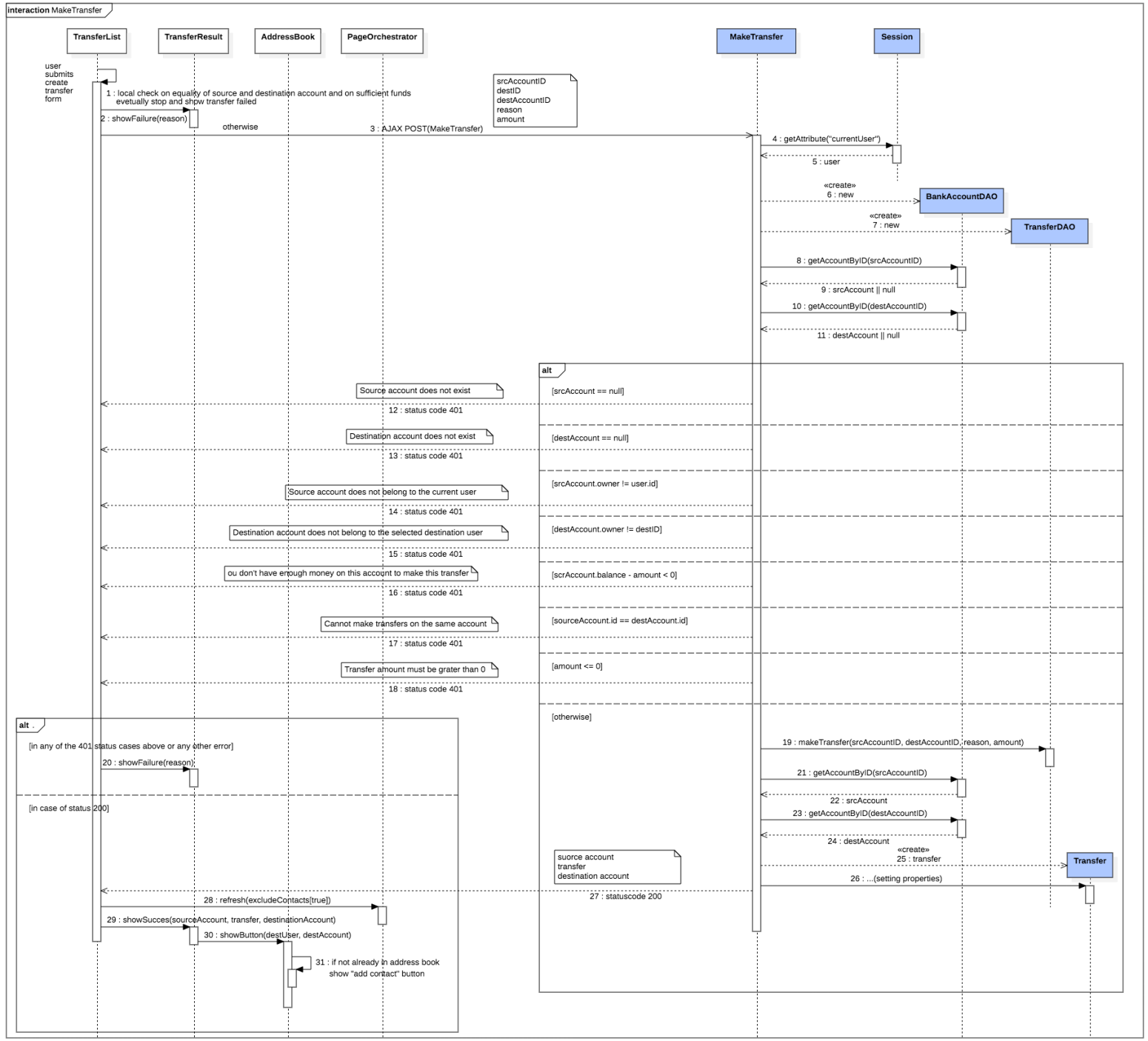

interaction Register
login.html & loginManagment.js
Register
Session
sessionStorage
Window
1 : check same passwords
click
submit
register
eventually report invalidity
2 : POST
«create»
3 : new
UserDAO
email
name
password
4 : getUserByEmail(email)
5 : user || null
alt
6 : status code 401
email already exists
[user != null]
7 : show error
[otherwise]
8 : registerUser(name,email,password)
9 : getUserByEmail(email)
10 : user
11 : setAttribute("currentUser", user)
user code and name
12 : status code 200
13 : set user name and code
14 : location.href = home.html

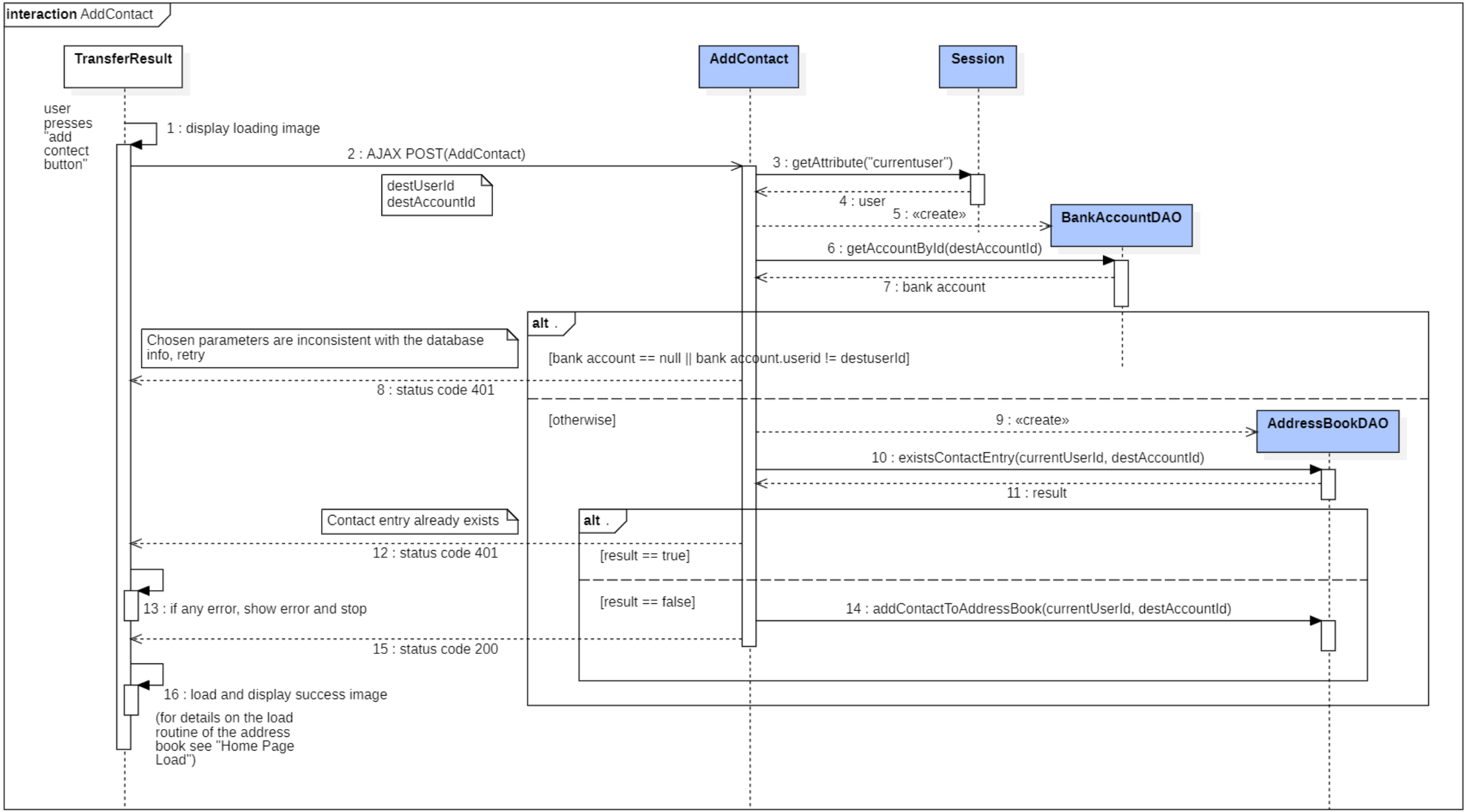

interaction Logout
UserInfo
sessionStorage
Logout
Session
login.html
user clicks on header logout button
1 : clear
2 : GET(Logout)
3 : invalidate
4 : redirect

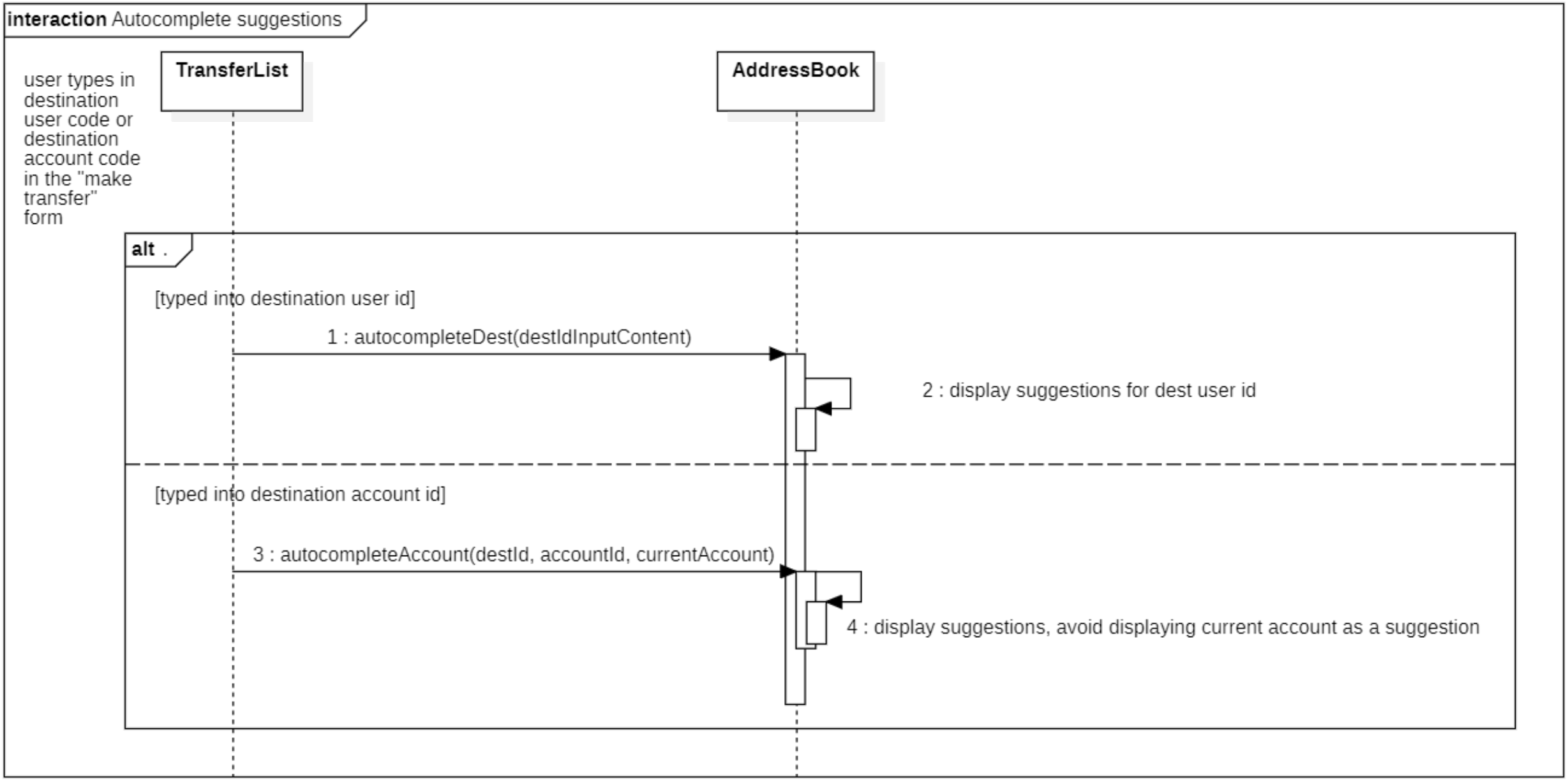

interaction Home page load
home.html & bank.js
sessionStorage
GetAccounts
Session
GetContacts
on load
«create»
1 : new Page Orchestrator
Page Orchestrator
2 : start
«create»
3 :
User Info
«create»
4 :
AccountList
«create»
5 :
TransferList
«create»
6 :
TransferResult
«create»
7 :
AddressBook
8 : refresh
9 : show
10 : get user name and code
11 : user name and code
12 : show
13 : AJAX GET(GetAccounts)
14 : getAttribute("currentUser")
15 : user
«create»
16 : new
BankAccountDAO
17 : getAccountsByUserID(user.id)
18 : accounts
user accounts
19 : status code 200
20 : update
opt if an account was already selected
21 : auto click the account
22 : show(accountId)
if (!excludeContacts)
23 : load
24 : AJAX GET(GetContacts)
25 : get User
26 : user
«create»
27 :
AddressBookDao
28 : getAddressBookById(userId)
29 : addressBook
user contacts
30 : status code 200
31 : store contacts

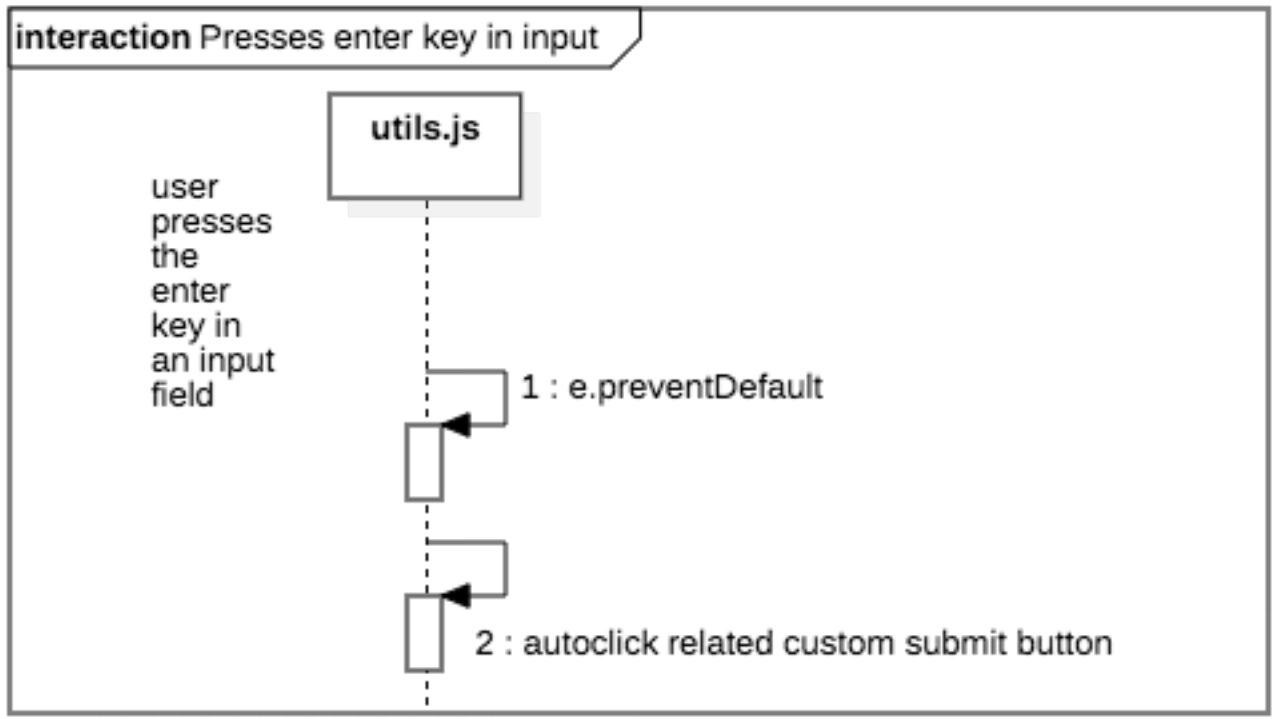

interaction Creation of Account
hom.html & bank.js
AccountList
CreateAccount
Session
user presses button to create account
1 : click event
2 : local check if name already exists and eventually display error and stop
account name
3 : AJAX POST(CreateAccount)
4 : getAttribute("currentUser")
5 : user
«create»
6 : new
BankAccountDAO
7 : getAccountByUserIDAndName(user.id)
8 : account || null
alt
[account != null]
"chosen account name already exists"
9 : status cose 401
10 : show error
[otherwise]
11 : createAccount(user.id, name)
12 : status code 200
13 : show
(for details on the show routine see "Home Page Load")


interaction Selection of an account
AccountList
TranferList
GetAccountDetails
Session
user opens an account to get the transfers
1 : manage open buttons
2 : show(accountId)
3 : AJAX GET(GetAccountDetails?accountId)
4 : getAttribute("currentUser")
5 : user
«create»
6 : new
BankAccountDAO
7 : getAccountByID(account_id)
8 : account || null
alt
[account == null || account.owner != user.id]
account does not exist or is not yours
9 : status code 401
10 : showError
[otherwise]
«create»
11 : new
TransferDAO
12 : getTransfersByAccountId(account.id)
13 : transfers
transfers
14 : status code 200
15 : update


interaction MakeTransfer
TransferList
TransferResult
AddressBook
PageOrchestrator
MakeTransfer
Session
BankAccountDAO
TransferDAO
Transfer
user submits create transfer form
1 : local check on equality of source and destination account and on sufficient funds evetually stop and show transfer failed
2 : showFailure(reason)
otherwise
3 : AJAX POST(MakeTransfer)
srcAccountID
destID
destAccountID
reason
amount
4 : getAttribute("currentUser")
5 : user
«create»
6 : new
«create»
7 : new
8 : getAccountByID(srcAccountID)
9 : srcAccount || null
10 : getAccountByID(destAccountID)
11 : destAccount || null
alt
[srcAccount == null]
Source account does not exist
12 : status code 401
[destAccount == null]
Destination account does not exist
13 : status code 401
[srcAccount.owner != user.id]
Source account does not belong to the current user
14 : status code 401
[destAccount.owner != destID]
Destination account does not belong to the selected destination user
15 : status code 401
[scrAccount.balance - amount < 0]
ou don't have enough money on this account to make this transfer
16 : status code 401
[sourceAccount.id == destAccount.id]
Cannot make transfers on the same account
17 : status code 401
[amount <= 0]
Transfer amount must be grater than 0
18 : status code 401
[otherwise]
19 : makeTransfer(srcAccountID, destAccountID, reason, amount)
21 : getAccountByID(srcAccountID)
22 : srcAccount
23 : getAccountByID(destAccountID)
24 : destAccount
«create»
25 : transfer
suorce account
transfer
destination account
26 : ...(setting properties)
27 : statuscode 200
alt .
[in any of the 401 status cases above or any other error]
20 : showFailure(reason)
[in case of status 200]
28 : refresh(excludeContacts[true])
29 : showSucces(sourceAccount, transfer, destinationAccount)
30 : showButton(destUser, destAccount)
31 : if not already in address book
show "add contact" button


interaction AddContact
TransferResult
AddContact
Session
BankAccountDAO
AddressBookDAO
user presses "add contect button"
1 : display loading image
2 : AJAX POST(AddContact)
destUserId
destAccountId
3 : getAttribute("currentuser")
4 : user
5 : «create»
6 : getAccountById(destAccountId)
7 : bank account
alt .
Chosen parameters are inconsistent with the database info, retry
[bank account == null || bank account.userid != destuserId]
8 : status code 401
[otherwise]
9 : «create»
10 : existsContactEntry(currentUserId, destAccountId)
11 : result
Contact entry already exists
alt .
12 : status code 401
[result == true]
[result == false]
13 : if any error, show error and stop
14 : addContactToAddressBook(currentUserId, destAccountId)
15 : status code 200
16 : load and display success image
(for details on the load routine of the address book see "Home Page Load")


interaction Autocomplete suggestions
user types in destination user code or destination account code in the "make transfer" form
TransferList
AddressBook
alt .
[typed into destination user id]
1 : autocompleteDest(destIdInputContent)
2 : display suggestions for dest user id
[typed into destination account id]
3 : autocompleteAccount(destId, accountId, currentAccount)
4 : display suggestions, avoid displaying current account as a suggestion


interaction Presses enter key in input
utils.js
user presses the enter key in an input field
1 : e.preventDefault
2 : autoclick related custom submit button