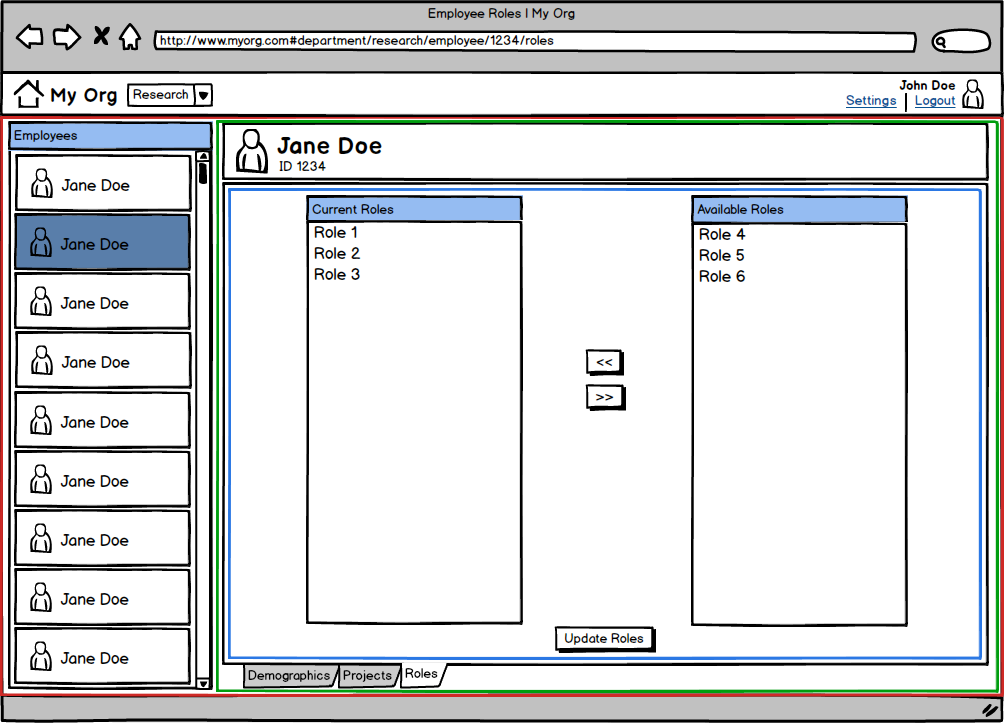
Navigation Flow

The purpose of this document is to walk through the navigation flow that takes place inside the router and to explain a bit of the how and why of this process.

To begin with, it should be noted that this is not a trivial problem. The complexity arises due to the representation of the application as a sort of hierarchical state machine. At any time any part of this hierarchy can transition to a new state and when this happens, in order to account for the common needs of application developers, a strict lifecycle must be maintained, governing the actual transitions. To further complicate matters, each phase of the navigation process must be asynchronous in order to handle such common needs as loading new components, displaying nag dialogs to users or even calling the server to authorize user access or load data.

# Scenarios

In order to make this explanation more concrete, we can use a simple example of an application with three levels of routers.



This mock app is used by "My Org" to manage basic employee information. The app has three main areas (level one - red): home view, settings view and department view. Within the department view there is a list of that department's employees. You can select an employee to bring up an employee detail view (level 2 - green). Within that detail view there are three tabs: demographics, projects and roles (level 3 - blue). The screenshot below shows the state of the app when the "research" department is being viewed, employee "Jane Doe" is selected and we are editing her roles.

Now let's consider several navigation scenarios, each beginning with the application in the state shown in the picture above:

1. Given that the user has changed the roles for this employee, but not yet clicked "Update Roles", if the user clicks on the "Projects" tab we may need to show a nag dialog informing the user that they have uncommitted changes (an async action in the middle of navigation). The user can then choose to continue and discard changes or to cancel (stopping navigation entirely). Here is an important detail: the same thing should happen if they select a different employee (level 2), a different department (level 1) or a different area of the application (level 1).
2. Given that the user has changed the roles for this employee and has clicked "Update Roles" but the update is still processing, if the user clicks on the "Projects" tab we must cancel navigation so that the user can receive the confirmation message from the backend. The same details apply with regards to changes in the hierarchy as mentioned above.
3. Given that the user has not changed anything in the roles screen and clicks on the "Projects" tab (or has clicked continue in the dialog from #1) we need to perform a state transition from department/research/employee/1234/roles to department/research/employee/1234/projects The router should not change the state at levels 1 or 2. It should only change the state of level 3. The above levels should not perceive the state change since this is encapsulated inside the employee controller.
4. Given that the user has not changed anything in the roles screen and clicks on a different employee (or has clicked continue in the dialog from #1) we need to perform a state transition from department/research/employee/1234/roles to department/research/employee/4321/demographics This is a change in level 2 and 3. The department screen remains the same. However, it is more interesting because it is only the id of the employee that changes. Before we can make this transition, we need to make sure we can transition away from the current employee. Then we can transition to the new employee, but we may want to re-use the employee screen instance and just update it with the new id. Or perhaps we want to throw out the instance and create a new one? This second option is less efficient, but the developer may have an external need to do it this way.
5. Pertaining to #4, imagine that employees can only view other employees whose "employee level" is lower than their own. Suppose that Joe knows the employee id of his manager and types that into the browser. The employee screen (level 2) should be able to deny access before the new screen is shown based on its own internal logic. It may also be convenient to abstract this so that it happens automatically as part of the navigation process.
6. Given that the user has not changed anything in the roles screen and selects a different department from the drop down list (or has clicked continue in the dialog from #1) we need to perform a state transition from department/research/employee/1234/roles to department/accounting. This is a transition of all 3 levels. Levels 3,2 and 1 all need to be queried to see if the transition can take place. Again though, this is only a change in the level 1 parameter to "accounting". We may wish to keep the department screen instance and just re-use it with a new set of inputs....or not.
7. Given that the user has not changed anything in the roles screen and selects settings (next to the avatar). This is also a transition in all three levels and is subject to all the transition queries mentioned above. However, this results in the discard of the department screen which is replaced with a new settings screen. The settings screen may have its own internal router to manage the navigation between different categories of settings such as: account, preferences, demographics, etc. Upon navigating to this screen a 2nd level transition may need to occur internally.
8. The same 3 level transition as mentioned in #6 would also occur when clicking on the home icon or on the logout link. The home screen may itself be a sort of dashboard with multiple different router zones (sibling routers rather than child routers).
9. Given a home screen with multiple sibling routers, if the user navigates back to our employee roles screen (perhaps by clicking the back button) then we will need to check all siblings to see if we can navigate away. Bear in mind that each sibling could also have child routers. These need to be checked from the inside out (most specific area of user focus to most general).

There are many more scenarios one can imagine from the screenshot. These are just a few selected to show some variation and potential complexity.

# Lifecycle

As described above, each controller in the hierarchy needs to have the ability to allow or disallow navigation to or from itself. Additionally, this process may need to involve asynchronous code. Thus the controller can opt into the activation lifecycle by implementing special callbacks:

* **canDeactivate**: The controller can return a boolean or a promise of a boolean to cancel navigation away from itself.
* **canActivate**: The controller can return a boolean or a promise of a boolean to cancel navigation to itself.
* **deactivate**: Performs cleanup after navigation has been permitted.
* **activate**: Performs setup after navigation has been permitted.

It is important to separate the query (canActivate, canDeactivate) callbacks from the command callbacks (activate, deactivate). Without this, hierarchies won't work. See below for the explanation of the activation flow.

*Note that canActivate and activate receive the input parameters which were extracted from the URL. This data may have been altered by a model binder (see below).*

# Flow

In order to handle the diverse set of possibilities, I've recommended a strict activation lifecycle. This was implemented in the existing prototype, but some flaws were discovered with regards to hierarchical queries. Because of the way the lifecycle was processed it was not possible to handle correctly and consistently all the scenarios described above. Previously, some callbacks were processed in the opposite order from what they should be. Others would be processed two or three times, instead of one. These flaws were discovered during my work on Durandal's router, independent of this project. It was not possible to fix Durandal's current design without introducing massive breaking changes. The insight gained from that is being used to re-vamp the internal processing of the navigation flow. Below is a brief explanation of my thinking in terms of navigation request handling, picking up from the point in time immediately after the navigation request is dequeued internally:

1. Build Navigation Plan

* Construct a plan by walking the existing hierarchy and comparing it to the specified components for the new route request.The following rules will be used to determine what should happen to the existing component instance.
  + If the route fragment resolves to the same component:
    - If the route input parameters are the same: reuse component instance, skip lifecycle (NO\_CHANGE)
    - If the route input parameters are different: reuse component instance, invoke lifecycle (INVOKE\_LIFECYCLE)
    - If the previous component implements the developer hook `determineActivationStrategy`, call to let the developer specify the activation strategy for this instance: REPLACE, INVOKE\_LIFECYCLE or NO\_CHANGE.
  + If the route fragment resolves to a different component:
    - Dump previous component instance (REPLACE)
  + The previous rules will be applied for each sibling route as well.
  + If the component instance is being reused, then
    - If the reused instance has NO\_CHANGE, then use the above rules to evaluate the child components.
    - If the reused instance has INVOKE\_LIFECYCLE, then evaluate the the child components, but all children must evaluate to at least INVOKE\_LIFECYCLE. They cannot be NO\_CHANGE.

1. Can Deactivate Previous States?

* Once a plan is constructed, we walk the plan tree invoking the `canDeactivate` callback on each node that is marked as REPLACE (previous instance) or INVOKE\_LIFECYCLE (existing instance). We should run callbacks from the inside out, representing the most specific user context to the most general.
* If all these evaluate to true (or a promise of true) we proceed to the next step.

1. Load New Components

* Any element in the tree that is marked with REPLACE should have the replacement component loaded.
* New components should be checked to see if they have child routers. If they do, we need to extend the plan to include the new component's specifications and then load the child components.
* If all component loading succeeds, proceed to the next step.

1. Run Model Binders

* Inspect the input parameters for the route. Do we have a model binder that matches any parameters? Some examples:
  + Convert an id into a loaded entity
  + Convert a string that represents a number into an actual number
* After all model binders have run (async), move to the next step.

1. Can Activate Next State?

* Walk the plan tree invoking `canActivate` on each node that is marked as REPLACE (new instance) or INVOKE\_LIFECYCLE (existing instance). Pass the result of the model-bound parameters into this callback for use by the developer. This should occur from the outside in, representing the most general user context to the most specific.
* If all these evaluate to true (or a promise of true) we proceed to the next step.

1. Deactivate Previous State

* At this point we know we have permission to make the state transition. So, we run the `deactivate` callbacks on on each node that is marked as REPLACE (previous instance) or INVOKE\_LIFECYCLE (existing instance). We should run callbacks from the inside out, representing the most specific user context to the most general.
* If no errors occur, we move on to the next step.

1. Activate Next State

* Run the `activate` callbacks on on each node that is marked as REPLACE (new instance) or INVOKE\_LIFECYCLE (existing instance). Pass the result of the model-bound parameters into this callback for use by the developer. This should occur from the outside in, representing the most general user context to the most specific.
* If no errors occur, we move on to the next step.

1. Finalize Navigation

* Refresh navigation models based on state changes
* Refresh bound hrefs based on state changes
* Mark the router as no longer navigating

After this process completes, the router will check its internal request queue to see if another navigation request has been made while the previous request was being processed.

*Note: If a new request comes in during the async processing of an existing request, we may be able to short-circuit the existing operation by checking for this case between pipeline steps. This could be represented by a special step. That said, once we reach step 6, we can no longer cancel without causing inconsistencies in state.*

Another important note is that the above steps are represented by an asynchronous pipeline. The pipeline is built by a pipeline provider, which is injected into the root router. This allows the developer to customize the pipeline by creating their own provider and configuring it with the steps they want. For example, they could remove model binders or the canActivate step. Or they could add a custom step that performs per-route user authorization.

If the transition fails or is canceled for any reason, the browser url is reverted to its previous state using history replace.

*Note: Any step in the pipeline can cancel navigation or redirect.*