Router Refactorings

TODOs

- Describe API of how it will work with Tobias change
- Use cases: Describe resolve
- Investigate the matchers
- Better understand many different ways to get hold of query params.

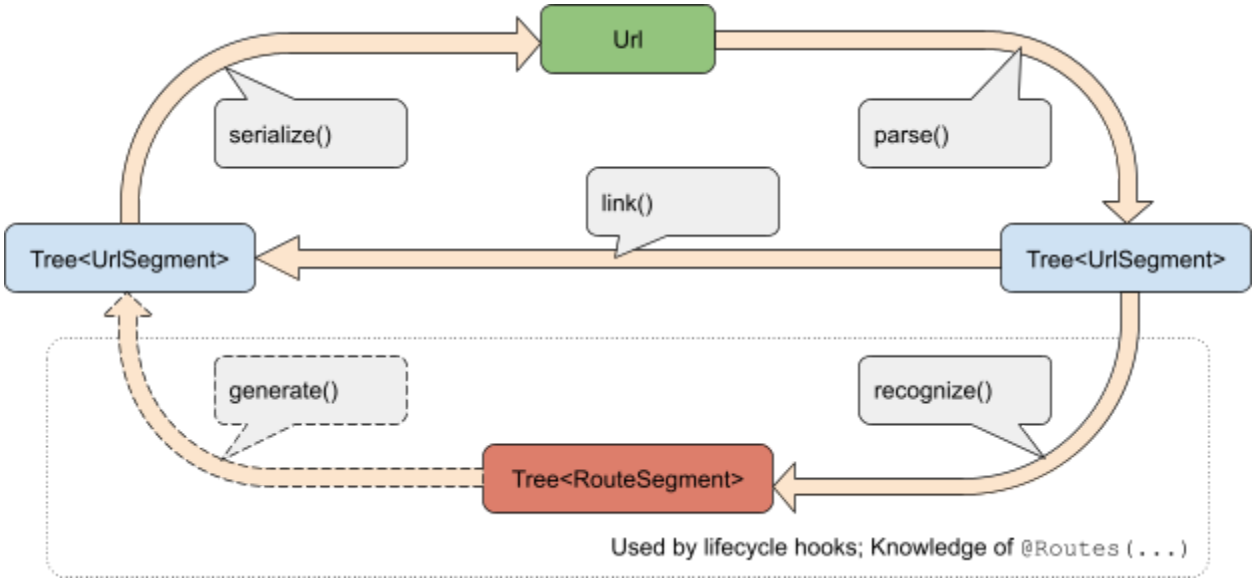
Refactoring Plan

1. Merge Instruction and ComponentInstruction
2. Refactor Route
    - Remove AsyncRoute
    - Change component:(Type|string|Promise<Type>)
    - Add ComponentResolver
    - Change aux to outlet
3. Remove canActivate; Add canActiveteChild
4. Refactor lifecycle hooks to bubble events all the way to the top.
5. Link Generation

Instruction, URL, Redesign

Goals:

- Creating route links needs to be efficient(synchronous) (can not depend on loading `@Routes`)
- The mental model / data flow needs to be easy to explain



Classes:

- `Url`: A serialized / string representation of the Url. (This is what is in the browser URL bar)
- `UrlSegment`: A `Url` can be broken down to `UrlSegment` using `parse()` method. A `UrlSegment` can be created without knowing anything about the `Routes` which are present in the application.
- `RouteSegment`: A `RouteSegment` is created from one or more `UrlSegment` by recognizing the `Routes`. The `recognize()` operation is slow since each step is asynchronous (lazy loading of routes).
- `Tree<T>`: A segment (`UrlSegment` or `RouteSegment`) forms a `Tree` (primary and named/auxiliary routes) The `Tree` provides parent/child relationships between the segments. Segments themselves are 1) immutable and 2) can belong to more than one `Tree` and hence 3) a child segment may be different depending on which tree we are part of. Most URLs are just lists as in `/admin/profile`. Such lists are a special case of a tree with a single child. However named/auxiliary routes allows for branching of trees. For example: `/admin(/chat)/profile(/changes)`.

```
/ -----> admin -----> profile
    \--> chat    \--> changes
```

    - Trees have navigation API such ups `parent()`, `child()`, `sibling()`, etc...
    - Tree segments can be checked with identity with `===`. (For example to see if navigation events from/to have parts which are staying the same)

Methods:

- `parse()`: An act of parsing a `Url` into a `Tree<UrlSegment>`. This operation is fully synchronous (hence fast) and can be performed without the need to refer to

`@Routes()`. Parsing/serializing is pluggable by the application developer so that they can control the URL formatting.

- `link()`: A link allows the transformation of a source `Tree<UrlSegment>` into a destination `Tree<UrlSegment>`. The transformation represent a navigation intent (Source tree of `/admin` and would like to navigate to `profile` hence the resulting tree is `/admin/profile`).
- `serialize()`: An act of turning a `Tree<UrlSegment>` into a `Url`.This operation is fully synchronous (hence fast) and can be performed without the need to refer to `@Routes()`. Parsing/serializing is pluggable by the application developer so that they can control the URL formatting.
- `recognize()`: An act of turning a `Tree<UrlSegment>` into `Tree<RouteSegment>`. The transformation is asynchronous and requires lazy loading of routes.
- `generate()`: An act of turning a `Tree<RouteSegment>` into `Tree<UrlSegment>`. This is a rare operation. Only used when route changes are triggered programmatically.

```
interface Tree<T> {
  root: T;
  current: T;
  parent(node: T): T|null;
  firstChild(node: T): T|null;
  nextSibling(node: T): T|null;
  previousSibling(node: T): T|null;
}

class UrlSegment { // immutable class
  outlet: string;
  segment: string;
  parameters: {[key:string]:string}
}

class RouteSegment { // immutable class
  router: Router;
  outlet: string;
  parameters: {[key:string]:string}
  urlSegments: UrlSegments[];
  componentType: Type;
  component: any;
}
```

Simplify Instruction (Obsolete see redesign)

Currently the Router has `Instruction` and `ComponentInstruction`. Here:

- `Instruction`: This represents the complete route to navigate to, so it should be a called `Route`, but that conflicts with `Route` annotation. So let's just keep the name.
- `ComponentInstruction`: This is an isolated information only visible to a given component. While I understand that it encapsulates information to a specific component, in practice this is over constraining the information. Instead I propose that we merge the information into `Instruction`.

```
class Instruction {
  id: any; // two instructions are same if their IDs match.
  outlet: string;
  rule: Rule;
  query: RouteParams;
  data: any;
  router: Router;
  isTerminal: boolean;
  isResolved: boolean;

  parent: Instruction;
  /*
   * [mutable] since we will have `null` during `canActivate`
   * since we have not loaded/materialized the child route.
   * In children the first child is the primary route.
   */
  children: Instruction[];

  /*
   * [mutable] since we will have `null` during `canActivate`
   * since we have not loaded/materialized the child route.
   */
  component: any;
}
```

Merge AUX Route into just Routes

There are situations where a component may wish to have more than one route-outlet. For example showing two views side by side. In such a case one view is primary and the other view is secondary (auxiliary). Example: Norton Commander

Instead of treating primary route and AUX routes as different, I think we can just say that at each level there can be any number of child routes. We can then say that the first route is special in the sense that it can be unnamed.

This means that we need to rename `aux` to `outlet`.

```
@Component({
  template: `
    <route-outlet></route-outlet>
    <route-outlet name="right"></route-outlet>
`
})
@Routes([
  {path: '/home', name: 'HomeCmp', outlet: 'primary' }
  {path: '/detail, name: 'DetailCmp', outlet: 'primary' }
  {path: '/modal', name: 'ModalCmp', outlet: 'right' }
  {path: '/confirmation', name: 'ConfirmationCmp', outlet: 'right'
}
  {path: '/redir', redirect: [], outlet: 'right' }
])
```

In the above example any set of routes which share the same outlet name are exclusive.

Router Link Behavior

```
['/app', 'HomeCmp']
['/app', [['HomeCmp']] ] // same as above
['/app', [['HomeCmp'], ['ConfirmationCmp']] ]
['/app', 'ModelCmp'] // navigate to child route in the right outlet

['/app', '!right'] // remove right outlet
```

Links

Generating links is a key component of web-applications. Most link generation is relative to the current route. Link generation must be done on the URL level (as opposed to on the Route level) because we need to be able to generate links to routes which have not been loaded yet, and we can't expect that links would cause lazy loading of routes.

We can think of a link generation as mutation to the current route. We start we the `Tree<UrlSegment>` and then apply a set of transformations (the link DSL) to the `Tree<UrlSegment>` to produce a new `Tree<UrlSegment>`.

For simplification there are two special routes.

1. `""`(unnamed) route: This is the primary route and used in most cases.
2. `"aux"`: This is a special secondary route whose name can be deduced from the URL, and hence it does not have to be present in the URL.

Examples:

- `/user/profile`: This is a primary route with an unnamed outlet.
  ```
  :/ ---> :user ---> :profile
  ```
- `/user(/notes)/profile`: Same as above plus `/notes` in the `aux` outlet (`"aux"` implied).
  ```
  :/ ---> :user ---> :profile
      \-> aux:notes
  ```
- `/user(/notes//right:/chat)/profile`: Same as above plus `/chat` in the `right` outlet.
  ```
  :/ ---> :user ---> :profile
      |-> aux:notes
      \-> right:chat
  ```

Mutating Links

All of the below examples assume that we have the following current route and that the mutations are competed from the highlighted node.

```
/admin/user/profile(right:/chat)/detail

:/ ---> :admin ---> :user ---> :profile ---> :detail
                            \-> right:chat
```

Examples ([link] => result):

- `['/home']` (or `['../home'']`) => `/home`
- `['settings']` (or `['./settings']`) => `/admin/user/settings(right:/chat)`
- `['right:']` (no path, hence close) => `/admin/user/profile/detail`
- `['aux:notes']` => `/admin/user/profile(/notes//right:chat)/detail`
- `['..' [['settings'], ['aux:notes']]]` => `/settings(/notes)`
- `['/settings', '/aux:notes']` => `/settings(/notes)`
- `[{a:123}]` => `/admin/user;a=123(right:/chat)/detail`
- `['/', {a:123}]` => `/admin/user/profile(right:/chat)/detail?a=123`

Notes:

- Root route gets query parameters all others get matrix parameters. (Since serialization is user overridable, a user can write their own serializer which can chose to do ether things.)

Move `@CanActivate` to `canActivateChild`

`canActivate` needs to be injectable. It also needs to be able to prevent navigation to lazy loaded routes, therefore it can not be in the lazy loaded root. The simplest way forward is to rename it to `canActivateChild` and move it to the parent component. The mental model is that parent decides if a child route can be activated.

Order of LifeCycle Events

Upon navigation to a new Route, Router needs to deactivate the old Route and activate the new Route. Each event may return a promise which will delay the next event from running until the promise is resolved.

Assume we go from `oldInstruction=/a/b1/c` to `newInstruction=/a/b2/d`. Here the `a` will get reused.

1. `oldC.canDeactivate(newInstruction, oldInstruction)`: We need to start from leaf
2. `oldB1.canDeactivate(newInstruction, oldInstruction)`:
3. `sameA.canDeactivate(newInstruction, oldInstruction)`:
4. `root.canDeactivate(newInstruction, oldInstruction)`: At this point everyone voted and agreed that we can deactivate.
5. `root.canActivateChild(newInstruction, oldInstruction)`: We need to start from root
6. `sameA.canActivateChild(newInstruction, oldInstruction)`:

7. `newB2.canActivateChild(newInstruction, oldInstruction)`: At this point everyone voted that we can activate.
8. `newB2.onActivate(newInstruction, oldInstruction)`:
9. `newD.onActivate(newInstruction, oldInstruction)`:
10. <route-outlet> switches views from the old route to the new route as well as update the URL at this time.
11. `oldC.onDeactivate(newInstruction, oldInstruction)`:
12. `oldB1.onDeactivate(newInstruction, oldInstruction)`:

TODO: describe how we can use onActivate to resolve data loads.

Link Generation

See the alternate proposal further down in this document

There is an outstanding issue that we can not create a link to a route until that route is lazy loaded (Because the lazy loaded route has the URL information). Because routes can be programmatically declared Specifically:

```
<a [routerLink]="['/Admin', 'User', 'Detail']">link to user details</a>
```

The above path can not be resolved until each section of the path is loaded.

Here are potential solutions:

1. Don't allow linking deeper than the currently declared route. (X: there are many use case examples where this is needed.)
2. Eagerly lazy load the route information when creating a link to a deep route. (X: seems like a performance problem)
3. Generate a route map offline and load it at application startup. See: Deep Linking in Template proposal. (X: There will always be a case where a complete route list can not be generated offline because of programmatically declared routes)
4. Have an unresolved form of the link which then redirects once the link is visited and all of the child routes are lazy-loaded. (Let's explore this more in depth)

It looks like the we need to deal with unresolved routes. Let's define:

- A canonical route URL is the one which the the developer wants the user to see. `http://server/admin/user/37/details`
- An unresolved route URL is the one which the router can not resolve due to lack of routing information. An unresolved URL has an easily recognizable format for our parser.

Something like this:
`http://server/admin/:User;id=37/:Detail`

The router would then have to be intelligent enough to redirect an unresolved route to canonical route. In practice this would not be an issue for majority of cases.

- For vast majority of links this is not an issue. It only comes into play if we want to deep link into a child route at least two levels deeper than the current route and which has not yet been lazy loaded.
- User clicks on unresolved route. The router starts lazy loading the components and properly resolves the route before the Browser URL is updated with a resolved URL. (User nevers sees unresolved URL)
- User “opens in a new tab” in which case the user of the application sees unresolved route, once the route is lazy loaded the browser history is replaced with the correct route. (User sees unresolved URL briefly while the application is loading, then the browser history is replaced with the correct URL).
    - It is only during this brief amount of time that it is possible for someone to grab an unresolved URL. Even if that happens, unless the developer renames the route, it would not be an issue.

Lazy Loading of Code

Problem: How do we succinctly specify how components are loaded
Solution:

1. Use the route name as the canonical name for the lazy loading
2. Have a `ComponentResolver` strategy to control the loading.

root_component.ts

```
import Comp form 'app/component/bar';

@Routes([
  { name: 'Child', path: '/child', component: Comp }
  { name: 'Child', path: '/child', component: `app/component/bar` }
  { name: 'Child', path: '/child', loader: () => require(...) }
])
class RootComponent { … }
```

child.ts

```
export default class ChildComponent { … }
```

`ComponentResolver`

```
interface ComponentResolver {
  resolve(path: string[]): Promise<HostViewFactory>;
}

class OfflineComponentResolver implements ComponentResolver {
  resolve(path: (Type|string)): Promise<HostViewFactory> {
    return System.import(path.join('/') + '-gen');
  }
}

class JITComponentResolver implements ComponentResolver {
  constructor(private compiler: Compiler) {}
  resolve(path: (Type|string)): Promise<HostViewFactory> {
    return System.import(path.join('/')).then((component: Type) =>
{
      return compiler.compile(component);
    });
  }
}
```

- We should also do `ComponentResolver` in `bootstrap()` so that we are consistent. `bootstrap(MyComponent) => bootstrap('/my-component');`
- This strategy will not work in Dart as there is no string based loader. To do deferred library loading there are two options:
    - Have `MapComponentResolver` where each library gets registered with a path and closure which does the lazy loading.
    - Allow the lazy loading closure to be inlined in the `const Route(lazy: () => aLib.load().then(() => aLib.MyComponent_HostViewFactory))` annotation.

Shorthand for Dart:

```
@Component({
  template: `
    <route-outlet></route-outlet>
`
})
@Routes([
  {path: '/home', name:'HomeCmp', resolve:() => myLib.loadLibrary }
  {path: '/detail, name: 'DetailCmp' }
])
```

NOTE:

- Get rid of `AsyncRoute`. All routes are now async.

Alternate proposal

Route configuration, link generation and lazy loading

In order to simplify lazy loading, we would drop the route names:

```
@Component({
  template: `
    <route-outlet></route-outlet>
    <route-outlet name=”right”></route-outlet>
`
})
@Routes([
  {path: '/home', outlet: ‘primary’ }
  {path: '/detail/:id’, outlet: ‘primary’ }
  {path: '/modal', outlet: ‘right’ }
  {path: '/confirmation', outlet: ‘right’ }
])
```

To create a link, we would need to specify all the segments:

```
<a [routerLink]=”[‘/home’]”>home</a>
<a [routerLink]=”[‘/detail’, detailId]”>home</a>
<!-- add non positional parameters to a route -->
<a [routerLink]=”[‘/detail’, detailId, {‘param’:
‘val’}]”>details</a>
```

The RouterLink directive would basically only join the parts with a “/”. Because all the URL parts are known even before a component is lazy loaded, we could generate a deep link even when the component is not loaded.

One caveat is that we could no more refer to a route by name and when changing the structure of an URL, both the route configuration and the RouterLink parameters should be updated.

Non-positional parameters

With the current design, the parameters of the root route are serialized in the query parameters and parameters for child routes are serialized as matrix parameters.
What we propose is:

- Query parameters become global and can be injected in any component (they are now shared and no more associated with a route any more),
- Each route could have non positional parameters, they would be serialized to matrix parameters and they could be injected in the corresponding component constructor.

To specify query parameters when generating a router link, add an hash as the first element of the array:

```
<a [routerLink]="[`{'shared': 'val'}`, '/detail', detailId,
{'local': 'val'}]">details</a>
```

Router Hooks

- All lifecycle hooks (`CanActivateChildren`, `OnActivate`, etc..) should also be available programmatically by registering callbacks with the `Router`.

NOTES

Aux route navigate away use-case

Imagine gmail-like UI, and we open a compose mole.

- Once the compose mole is open navigating back / forth keeps the mole open. This implies that navigation is for primary route only.

- Let’s write some text in the compose mole, and try to close it. This would ask the compose mole canDictivate. Now the mole would like to open a dialog asking for permission to throw away the draft. The dialog itself should be a aux route navigation event.
    - How do we navigate to a new route while navigation is going on?

Ideas:

- Primary route has a router
- Each aux has its own router which is a child of parent router.
- Once navigation is started the router at that level will not allow other navigation events, until the current one is resolved. This means that the compose mole would have to programatically open a discard dialog to ask the permission. (Not a route)

Suggestion for RouterLink representation

Propelling this suggestion are what I believe to be the five concepts that need representation

1. Path (a part of the URL)
2. Route Parameters
3. Query/Matrix Parameters
4. Name outlets
5. A tree of route segments targeting outlets

My goal is a representation that can encompass the most outlandish route but that degenerates into a graceful syntax for the most common cases.

I know we can't use the term but I'm going to describe the basic unit of this scheme as a "**route segment**" or `Segment` for short.

Here is a pseudo-code quasi-definition for the entire `Segment` model:

```
Segment:
  {
    '': { path: path, fork?: Segment }, // the unnamed outlet
    x:  { path: path, fork?: Segment }, // outlet named `x`
    y:  { path: path, fork?: Segment }  // outlet named `y`
  }

// string or array of (string|number|queryParams)
path: string | (string | number | queryParams)[]

queryParams: {} // a map
```

I think this is unambiguous

- An object on its own is a Segment
- The properties of a Segment are its target outlets, all siblings
- A string or array is a path
- A path array can be resolved into a string path
- The first element of a path array must be a string but remaining elements of a path array may appear in any order. The router assembles them into the string path in the manner discussed previously.

There are two very useful degenerate forms:

1. `RouterLink = string  // e.g., 'company/23/employee/12?foo=3&bar=bar'`
2. `RouterLink = [...]  // e.g., ['company', 23, 'employee', 12, {foo: 3, bar: 'bar'}]`

Each is easily resolved (internally … you'd never actually write this) into a segment of the form

```
  { '': { path: path }, fork: null } // segment with a single unnamed outlet
```

Where the path is either #1 or #2.

The moment a named route enters the picture, we must express the segment in its full form:

```
 { 'Alert': { path: 'alert' }, }
```

Here's a segment for the unnamed outlet that goes to customer #23, and then forks at the `EmployeeComponent` to its two outlets: unnamed and 'alert'. There are paths to both.

```
  { '': {
      path: ['customer', 23],
      fork: {
        '':      { path: [ 'employee', 12, {foo: 3,       bar: 'bar'}  ] },
        'alert': { path: [ 'alert',        {msg: 'Nutz?', mode: YesNo} ] }
      }
    }
  }
```

That's not something I'd want to paste into a template. I'd likely create it within my component and navigate to it. But I could put it into a template if you twisted my arm.

I do not know how we serialize these forms into an acceptable URL. But you all thought about this already.

Aside: what should the EmployeeComponent do if it doesn't have an `alert` outlet? Should we have a syntax for "*that path is optional*"?

```
'alert?': { path:  [ 'alert', {msg: 'Nutz?', mode: YesNo} ] }
'alert':  { path?: [ 'alert', {msg: 'Nutz?', mode: YesNo} ] }
'alert?': { path?: [ 'alert', {msg: 'Nutz?', mode: YesNo} ] }
```