After reading this guide, you'll know:

1. The role URLs play in a client-rendered app, and how it's different from a traditional server-rendered app.
2. How to define client and server routes for your app using Flow Router.
3. How to have your app display different content depending on the URL.
4. How to dynamically load application modules depending on the URL.
5. How to construct links to routes and go to routes programmatically.

Client-side Routing

In a web application, *routing* is the process of using URLs to drive the user interface (UI). URLs are a prominent feature in every single web browser, and have several main functions from the user's point of view:

1. **Bookmarking** - Users can bookmark URLs in their web browser to save content they want to come back to later.
2. **Sharing** - Users can share content with others by sending a link to a certain page.
3. **Navigation** - URLs are used to drive the web browser's back/forward functions.

In a traditional web application stack, where the server renders HTML one page at a time, the URL is the fundamental entry point for the user to access the application. Users navigate an application by clicking through URLs, which are sent to the server via HTTP, and the server responds appropriately via a server-side router.

In contrast, Meteor operates on the principle of *data on the wire*, where the server doesn't think in terms of URLs or HTML pages. The client application communicates with the server over DDP. Typically as an application loads, it initializes a series of *subscriptions* which fetch the data required to render the application. As the user interacts with the application, different subscriptions may load, but there's no technical need for URLs to be involved in this process - you could have a Meteor app where the URL never changes.

However, most of the user-facing features of URLs listed above are still relevant for typical Meteor applications. Since the server is not URL-driven, the URL becomes a useful representation of the client-side state the user is currently looking at. However, unlike in a server-rendered application, it does not need to describe the entirety of the user's current state; it needs to contain the parts that you want to be linkable. For example, the URL should contain any search filters applied on a page, but not necessarily the state of a dropdown menu or popup.

Using Flow Router

To add routing to your app, install the `ostrio:flow-router-extra` package:

```
meteor add ostrio:flow-router-extra
```

Flow Router is a community routing package for Meteor.

Using Flow Router Extra

Flow Router Extra is carefully extended `flow-router` package by `kadira` with waitOn and template context. Flow Router Extra shares original "Flow Router" API including flavoring for extra features like `waitOn`, template context and build in `.render()`. Note: `arillo:flow-router-helpers` and `zimme:active-route` already build into Flow Router Extra and updated to support latest Meteor release.

To add routing to your app, install the `ostrio:flow-router-extra` package:

```
meteor add ostrio:flow-router-extra
```

Defining a simple route

The basic purpose of a router is to match certain URLs and perform actions as a result. This all happens on the client side, in the app user's browser or mobile app container. Let's take an example from the Todos example app:

```
FlowRouter.route('/lists/:_id', {
  name: 'Lists.show',
  action(params, queryParams) {
    console.log("Looking at a list?");
  }
});
```

This route handler will run in two situations: if the page loads initially at a URL that matches the URL pattern, or if the URL changes to one that matches the pattern while the page is open. Note that, unlike in a server-side-rendered app, the URL can change without any additional requests to the server.

When the route is matched, the `action` method executes, and you can perform any actions you need to. The `name` property of the route is optional, but will let us refer to this route more conveniently later on.

URL pattern matching

Consider the following URL pattern, used in the code snippet above:

```
'/lists/:_id'
```

The above pattern will match certain URLs. You may notice that one segment of the URL is prefixed by `:` - this means that it is a *url parameter*, and will match any string that is present in that segment of the path. Flow Router will make that part of the URL available on the `params` property of the current route.

Additionally, the URL could contain an HTTP *query string* (the part after an optional `?`). If so, Flow Router will also split it up into named parameters, which it calls `queryParams`.

Here are some example URLs and the resulting `params` and `queryParams`:

URL	matches pattern?	params	queryParams
/	no		
/about	no		
/lists/	no		
/lists/eMtGij5AFESbTKfkT	yes	{ _id: "eMtGij5AFESbTKfkT"}	{ }
/lists/1	yes	{ _id: "1"}	{ }
/lists/1?todoSort=top	yes	{ _id: "1"}	{ todoSort: "top" }

Note that all of the values in `params` and `queryParams` are always strings since URLs don't have any way of encoding data types. For example, if you wanted a parameter to represent a number, you might need to use `parseInt(value, 10)` to convert it when you access it.

Accessing Route information

In addition to passing in the parameters as arguments to the `action` function on the route, Flow Router makes a variety of information available via (reactive and otherwise) functions on the global singleton `FlowRouter`. As the user navigates around your app, the values of these functions will change (reactively in some cases) correspondingly.

Like any other global singleton in your application (see the data loading for info about stores), it's best to limit your access to `FlowRouter`. That way the parts of your app will remain modular and more independent. In the case of `FlowRouter`, it's best to access it solely from the top of your component hierarchy, either in the "page" component, or the layouts that wrap it. Read more about accessing data in the UI article.

The current route

It's useful to access information about the current route in your code. Here are some reactive functions you can call:

- `FlowRouter.getRouteName()` gets the name of the route
- `FlowRouter.getParam(paramName)` returns the value of a single URL parameter
- `FlowRouter.getQueryParam(paramName)` returns the value of a single URL query parameter

In our example of the list page from the Todos app, we access the current list's id with `FlowRouter.getParam('_id')` (we'll see more on this below).

Highlighting the active route

One situation where it is sensible to access the global `FlowRouter` singleton to access the current route's information deeper in the component hierarchy is when rendering links via a navigation component. It's often required to highlight the "active" route in some way (this is the route or section of the site that the user is currently looking at).

In the Todos example app, we link to each list the user knows about in the `App_body` template:

```
{{#each list in lists}}
  <a class="list-todo {{activeListClass list}}">
    ...

    {{list.name}}
  </a>
{{/each}}
```

We can determine if the user is currently viewing the list with the `activeListClass` helper:

```
Template.App_body.helpers({
  activeListClass(list) {
    const active = ActiveRoute.name('Lists.show')
      && FlowRouter.getParam('_id') === list._id;

    return active && 'active';
  }
});
```

Rendering based on the route

Now we understand how to define routes and access information about the current route, we are in a position to do what you usually want to do when a user accesses a route---render a user interface to the screen that represents it.

In this section, we'll discuss how to render routes using Blaze as the UI engine. If you are building your app with React or Angular, you will end up with similar concepts but the code will be a bit different.

When using Flow Router, the simplest way to display different views on the page for different URLs is to use the complementary Blaze Layout package. First, make sure you have the Blaze Layout package installed:

```
meteor add kadira:blaze-layout
```

To use this package, we need to define a "layout" component. In the Todos example app, that component is called `App_body`:

```
<template name="App_body">
  ...
  {{> Template.dynamic template=main}}
  ...
</template>
```

(This is not the entire `App_body` component, but we highlight the most important part here). Here, we are using a Blaze feature called `Template.dynamic` to render a template which is attached to the `main` property of the data context. Using Blaze Layout, we can change that `main` property when a route is accessed.

We do that in the `action` function of our `Lists.show` route definition:

```
FlowRouter.route('/lists/:_id', {
  name: 'Lists.show',
  action() {
    BlazeLayout.render('App_body', {main: 'Lists_show_page'});
  }
});
```

What this means is that whenever a user visits a URL of the form `/lists/X`, the `Lists.show` route will kick in, triggering the `BlazeLayout` call to set the `main` property of the `App_body` component.

Components as pages

Notice that we called the component to be rendered `Lists_show_page` (rather than `Lists_show`). This indicates that this template is rendered directly by a Flow Router action and forms the 'top' of the rendering hierarchy for this URL.

The `Lists_show_page` template renders *without* arguments---it is this template's responsibility to collect information from the current route, and then pass this information down into its child templates. Correspondingly the `Lists_show_page` template is very tied to the route that rendered it, and so it needs to be a smart component. See the article on UI/UX for more about smart and reusable components.

It makes sense for a "page" smart component like `Lists_show_page` to:

1. Collect route information,
2. Subscribe to relevant subscriptions,
3. Fetch the data from those subscriptions, and
4. Pass that data into a sub-component.

In this case, the HTML template for `Lists_show_page` will look very simple, with most of the logic in the JavaScript code:

```
<template name="Lists_show_page">
  {{#each listId in listIdArray}}
    {{> Lists_show (listArgs listId)}}
  {{else}}
    {{> App_notFound}}
  {{/each}}
</template>
```

(The `{% raw %}{{#each listId in listIdArray}}{% endraw %}` is an animation technique for page to page transitions).

```
Template.Lists_show_page.helpers({
  // We use #each on an array of one item so that the "list" template is
  // removed and a new copy is added when changing lists, which is
  // important for animation purposes.
  listIdArray() {
    const instance = Template.instance();
    const listId = instance.getListId();
    return Lists.findOne(listId) ? [listId] : [];
  },
  listArgs(listId) {
    const instance = Template.instance();
    return {
      todosReady: instance.subscriptionsReady(),
      // We pass `list` (which contains the full list, with all fields, as a
function
      // because we want to control reactivity. When you check a todo item, the
      // `list.incompleteCount` changes. If we didn't do this the entire list would
      // re-render whenever you checked an item. By isolating the reactiviy on the
list
      // to the area that cares about it, we stop it from happening.
      list() {
        return Lists.findOne(listId);
      },
      // By finding the list with only the `_id` field set, we don't create a
dependency on the
      // `list.incompleteCount`, and avoid re-rendering the todos when it changes
      todos: Lists.findOne(listId, {fields: {_id: true}}).todos()
    };
  }
});
```

It's the `listShow` component (a reusable component) that actually handles the job of rendering the content of the page. As the page component is passing the arguments into the reusable component, it is able to be quite mechanical and the concerns of talking to the router and rendering the page have been separated.

Changing page when logged out

There are types of rendering logic that appear related to the route but which also seem related to user interface rendering. A classic example is authorization; for instance, you may want to render a login form for some subset of your pages if the user is not yet logged in.

It's best to keep all logic around what to render in the component hierarchy (i.e. the tree of rendered components). So this authorization should happen inside a component. Suppose we wanted to add this to the `Lists_show_page` we were looking at above. We could do something like:

```
<template name="Lists_show_page">
  {{#if currentUser}}
    {{#each listId in listIdArray}}
      {{> Lists_show (listArgs listId)}}
    {{else}}
      {{> App_notFound}}
    {{/each}}
  {{else}}
    Please log in to edit posts.
  {{/if}}
</template>
```

Of course, we might find that we need to share this functionality between multiple pages of our app that require access control. We can share functionality between templates by wrapping them in a wrapper "layout" component which includes the behavior we want.

You can create wrapper components by using the "template as block helper" ability of Blaze (see the Blaze Article). Here's how we could write an authorization template:

```
<template name="App_forceLoggedIn">
  {{#if currentUser}}
    {{> Template.contentBlock}}
  {{else}}
    Please log in see this page.
  {{/if}}
</template>
```

Once that template exists, we can wrap our `Lists_show_page`:

```
<template name="Lists_show_page">
  {{#App_forceLoggedIn}}
    {{#each listId in listIdArray}}
      {{> Lists_show (listArgs listId)}}
    {{else}}
      {{> App_notFound}}
    {{/each}}
```

```
  {{/App_forceLoggedIn}}
</template>
```

The main advantage of this approach is that it is immediately clear when viewing the `Lists_show_page` what behavior will occur when a user visits the page.

Multiple behaviors of this type can be composed by wrapping a template in multiple wrappers, or creating a meta-wrapper that combines multiple wrapper templates.

Changing Routes

Rendering an updated UI when a user reaches a new route is not that useful without giving the user some way to reach a new route! The simplest way is with the trusty `<a>` tag and a URL. You can generate the URLs yourself using helpers such as `FlowRouter.pathFor` to display a link to a certain route. For example, in the Todos example app, our nav links look like:

```
<a href="{{pathFor 'Lists.show' _id=list._id}}" title="{{list.name}}"
    class="list-todo {{activeListClass list}}">
```

Routing programmatically

In some cases you want to change routes based on user action outside of them clicking on a link. For instance, in the example app, when a user creates a new list, we want to route them to the list they just created. We do this by calling `FlowRouter.go()` once we know the id of the new list:

```
import { insert } from '../../api/lists/methods.js';

Template.App_body.events({
  'click .js-new-list'() {
    const listId = insert.call();
    FlowRouter.go('Lists.show', { _id: listId });
  }
});
```

You can also change only part of the URL if you want to, using the `FlowRouter.setParams()` and `FlowRouter.setQueryParams()`. For instance, if we were viewing one list and wanted to go to another, we could write:

```
FlowRouter.setParams({_id: newList._id});
```

Of course, calling `FlowRouter.go()`, will always work, so unless you are trying to optimize for a specific situation it's better to use that.

Storing data in the URL

As we discussed in the introduction, the URL is really a serialization of some part of the client-side state the user is looking at. Although parameters can only be strings, it's possible to convert any type of data to a string by serializing it.

In general if you want to store arbitrary serializable data in a URL param, you can use `EJSON.stringify()` to turn it into a string. You'll need to URL-encode the string using `encodeURIComponent` to remove any characters that have meaning in a URL:

```
FlowRouter.setQueryParams({data: encodeURIComponent(EJSON.stringify(data))});
```

You can then get the data back out of Flow Router using `EJSON.parse()`. Note that Flow Router does the URL decoding for you automatically:

```
const data = EJSON.parse(FlowRouter.getQueryParam('data'));
```

Redirecting

Sometimes, your users will end up on a page that isn't a good place for them to be. Maybe the data they were looking for has moved, maybe they were on an admin panel page and logged out, or maybe they just created a new object and you want them to end up on the page for the thing they just created.

Usually, we can redirect in response to a user's action by calling `FlowRouter.go()` and friends, like in our list creation example above, but if a user browses directly to a URL that doesn't exist, it's useful to know how to redirect immediately.

If a URL is out-of-date (sometimes you might change the URL scheme of an application), you can redirect inside the `action` function of the route:

```
FlowRouter.route('/old-list-route/:_id', {
  action(params) {
    FlowRouter.go('Lists.show', params);
  }
});
```

Redirecting dynamically

The above approach will only work for static redirects. However, sometimes you need to load some data to figure out where to redirect to. In this case you'll need to render part of the component hierarchy to subscribe to the data you need. For example, in the Todos example app, we want to make the root (`/`) route redirect to the first known list. To achieve this, we need to render a special `App_rootRedirector` route:

```
FlowRouter.route('/', {
  name: 'App.home',
  action() {
    BlazeLayout.render('App_body', {main: 'App_rootRedirector'});
  }
});
```

The `App_rootRedirector` component is rendered inside the `App_body` layout, which takes care of subscribing to the set of lists the user knows about *before* rendering its sub-component, and we are guaranteed there is at least one such list. This means that if the `App_rootRedirector` ends up being created, there'll be a list loaded, so we can do:

```
Template.App_rootRedirector.onCreated(function rootRedirectorOnCreated() {
  // We need to set a timeout here so that we don't redirect from inside a
redirection
  //   which is a limitation of the current version of FR.
  Meteor.setTimeout(() => {
    FlowRouter.go('Lists.show', Lists.findOne());
  });
});
```

If you need to wait on specific data that you aren't already subscribed to at creation time, you can use an `autorun` and `subscriptionsReady()` to wait on that subscription:

```
Template.App_rootRedirector.onCreated(function rootRedirectorOnCreated() {
  // If we needed to open this subscription here
  this.subscribe('lists.public');

  // Now we need to wait for the above subscription. We'll need the template to
  // render some kind of loading state while we wait, too.
  this.autorun(() => {
    if (this.subscriptionsReady()) {
      FlowRouter.go('Lists.show', Lists.findOne());
    }
  });
});
```

Redirecting after a user's action

Often, you just want to go to a new route programmatically when a user has completed a certain action. Above we saw a case (creating a new list) when we wanted to do it *optimistically*---i.e. before we hear back from the server that the Method succeeded. We can do this because we reasonably expect that the Method will succeed in almost all cases (see the UI/UX article for further discussion of this).

However, if we wanted to wait for the method to return from the server, we can put the redirection in the callback of the method:

```
Template.App_body.events({
  'click .js-new-list'() {
    lists.insert.call((err, listId) => {
      if (!err) {
        FlowRouter.go('Lists.show', { _id: listId });
      }
    });
  }
});
```

You will also want to show some kind of indication that the method is working in between their click of the button and the redirect completing. Don't forget to provide feedback if the method is returning an error.

Advanced Routing

Dynamically load modules

Dynamic imports was fist introduced in Meteor 1.5. This technique allows to dramatically reduce *Client's* bundle size, and load modules and dependencies dynamically upon request, in this case based on the current URI.

Assume we have `index.html` and `index.js` with code for `index` template and this is only place in the application where it depend on large `moment` package. This means `moment` package is not needed in the other parts of our app, and it will only waste bandwidth and slow load time.

```
<!-- /imports/client/index.html -->
<template name="index">
  <h1>Current time is: {{time}}</h1>
</template>
```

```
// /imports/client/index.js
import moment       from 'moment';
import { Template } from 'meteor/templating';
import './index.html';

Template.index.helpers({
  time() {
    return moment().format('LTS');
  }
});
```

```
// /imports/lib/routes.js
import { FlowRouter } from 'meteor/ostrio:flow-router-extra';

FlowRouter.route('/', {
  name: 'index',
  waitOn() {
    // Wait for index.js load over the wire
    return import('/imports/client/index.js');
  },
  action() {
    BlazeLayout.render('App_body', {main: 'index'});
  }
};
```

For more info and examples see this thread

Missing pages

If a user types an incorrect URL, chances are you want to show them some kind of amusing not-found page. There are actually two categories of not-found pages. The first is when the URL typed in doesn't match any of your route definitions. You can use `FlowRouter.notFound` to handle this:

```
// the App_notFound template is used for unknown routes and missing lists
FlowRouter.notFound = {
  action() {
```

```
    BlazeLayout.render('App_body', {main: 'App_notFound'});
  }
};
```

The second is when the URL is valid, but doesn't actually match any data. In this case, the URL matches a route, but once the route has successfully subscribed, it discovers there is no data. It usually makes sense in this case for the page component (which subscribes and fetches the data) to render a not-found template instead of the usual template for the page:

```
<template name="Lists_show_page">
  {{#each listId in listIdArray}}
    {{> Lists_show (listArgs listId)}}
  {{else}}
    {{> App_notFound}}
  {{/each}}
<template>
```

Analytics

It's common to want to know which pages of your app are most commonly visited, and where users are coming from. You can read about how to set up Flow Router based analytics in the Deployment Guide.

Server Side Routing

As we've discussed, Meteor is a framework for client rendered applications, but this doesn't always remove the requirement for server rendered routes. There are three main use cases for server-side routing.

Server Routing for API access

Although Meteor allows you to write low-level connect handlers to create any kind of API you like on the server-side, if all you want to do is create a RESTful version of your Methods and Publications, you can often use the `simple:rest` package to do this. See the Data Loading and Methods articles for more information.

If you need more control, you can use the comprehensive `nimble:restivus` package to create more or less whatever you need in whatever ontology you require.

Server Rendering

The Blaze UI library does not have support for server-side rendering, so it's not possible to render your pages on the server if you use Blaze. However, the React UI library does. This means it is possible to render HTML on the server if you use React as your rendering framework.

Although Flow Router can be used to render React components more or less as we've described above for Blaze, at the time of this writing Flow Router's support for SSR is still experimental. However, it's probably the best approach right now if you want to use SSR for Meteor.

Server Routing for additional resources

There might be additional resources that you want to make available on your server or receive web hooks. If you need anything more complicated with dynamic parts of the URL you might want to implement Picker which is a simple server-side router that handles dynamic routes.

If you need to authenticate the user when providing additional server-side resources such as PDF documents or XLSX spreadsheets, you can use `mhagmajer:server-router` package to do this easily. There is a blog article that

describes this in more detail.