Aurelia Cheat Sheet

A concise reference for Aurelia, covering core concepts, syntax, and best practices for building modern JavaScript applications.

Core Concepts

Components

Aurelia applications are built from components. Each component consists of a JavaScript view-model and an HTML view.

View-Model: The JavaScript class that manages the data and behavior for the view.
View: The HTML template that defines the structure and appearance of the component.

File naming convention: `my-component.js` (view-model) and `my-component.html` (view).

Data Binding

One-way binding (one-time)	`<div text.one-time="message"></div>` Value is set only once.
One-way binding (to-view)	`<div text.to-view="message"></div>` View is updated when the view-model property changes.
One-way binding (from-view)	`<input value.from-view="message">` View-model is updated when the view changes.
Two-way binding	`<input value.bind="message">` View and view-model stay in sync.
Delegate Binding	`<button click.delegate="submit()">Submit</button>` Attaches a delegated event listener.
Trigger Binding	`<button click.trigger="submit()">Submit</button>` Attaches a capturing event listener.

Dependency Injection

Aurelia uses Dependency Injection (DI) to manage dependencies between components. Dependencies are declared in the constructor of the view-model.

Example:

```
import { HttpClient } from 'aurelia-fetch-client';

export class MyComponent {
  static inject = [HttpClient];
  constructor(http) {
    this.http = http;
  }
}
```

Alternatively, use the `@inject` decorator (requires enabling decorators in TypeScript or Babel):

```
import { inject } from 'aurelia-framework';
import { HttpClient } from 'aurelia-fetch-client';

@inject(HttpClient)
export class MyComponent {
  constructor(http) {
    this.http = http;
  }
}
```

Templating

Basic Syntax

Aurelia's templating engine uses HTML enhanced with custom attributes and elements.

String interpolation: `${property}`

Attribute binding: `<div class.bind="condition ? 'class1' : 'class2'"></div>`

Event binding: `<button click.trigger="doSomething()">Click Me</button>`

Control Flow

`repeat.for`	Loops through a collection.
	`<div repeat.for="item of items">`
	`  ${item.name}`
	`</div>`

`if.bind`	Conditional rendering.
	`<div if.bind="condition">`
	`  This is shown when condition is true.`
	`</div>`

`with.bind`	Creates a binding context.
	`<div with.bind="user">`
	`  ${name} - ${age}`
	`</div>`

Value Converters

Value converters transform data for display in the view. Create a class with `toView` and optionally `fromView` methods.

Example:

```
export class UpperCaseValueConverter {
  toView(value) {
    return value.toUpperCase();
  }
}
```

Usage in view: `${message | uppercase}`

Custom Elements & Attributes

Creating Custom Elements

Custom elements allow you to create reusable UI components with encapsulated logic and presentation.

Define a class and associate it with an HTML template (view).

Example:

```
import { customElement } from 'aurelia-
framework';

@customElement('my-element')
export class MyElement {
  message = 'Hello, from my element!';
}
```

`my-element.html`:

```
<div>${message}</div>
```

Custom Attributes

Defining	Custom attributes allow you to extend HTML elements with custom behavior.
	`import { customAttribute } from 'aurelia-framework';`
	
	`@customAttribute('my-attribute')`
	`export class MyAttribute {`
	`  constructor(element) {`
	`    this.element = element;`
	`  }`
	
	`  valueChanged(newValue, oldValue) {`
	`    this.element.setAttribute('data-value', newValue);`
	`  }`
	`}`

Usage	`<div my-attribute="someValue"></div>`

Lifecycle Hooks

Aurelia provides lifecycle hooks that allow you to execute code at specific stages of a component's lifecycle.

- `constructor()`: Invoked when the component is created.
- `bind(bindingContext, overrideContext)`: Invoked when the binding context is established.
- `attach()`: Invoked when the component is attached to the DOM.
- `attached()`: Invoked after the component is attached to the DOM.
- `detached()`: Invoked when the component is detached from the DOM.
- `unbind()`: Invoked when the binding context is unbound.

Routing

Basic Configuration

Aurelia's router enables navigation between different views within your application.

Configure the router in your app's `configureRouter` method.

```
import { PLATFORM } from 'aurelia-pal';

export class App {
  configureRouter(config, router) {
    this.router = router;
    config.title = 'Aurelia App';
    config.map([
      { route: ['', 'home'], name: 'home',
moduleId: PLATFORM.moduleName('home/home')},
      { route: 'users', name: 'users',
moduleId: PLATFORM.moduleName('users/users') }
    ]);
  }
}
```

Add `<router-view>` in your main app view to display the routed content.

Navigation

`router.navigateToRoute(routeName, params)`	Navigates to a named route with optional parameters.
`router.navigate(url)`	Navigates to a specific URL.
`<a route-href="route: routeName; params.bind: { id: id }">Link</a>`	Creates a navigation link.

Route Parameters

Access route parameters in your view-model.

```
import { inject } from 'aurelia-framework';
import { ActivatedRoute } from 'aurelia-
router';

@inject(ActivatedRoute)
export class UserDetails {
  constructor(route) {
    this.route = route;
  }

  activate(params) {
    this.userId = params.id;
  }
}
```