NativeScript Cheatsheet

A quick reference guide for NativeScript, covering core concepts, UI elements, data binding, and common tasks.

Core Concepts

Application Structure

NativeScript applications are structured with a `app` directory at the root. This directory contains the application's core components.

Key files and directories include:

- `app.ts` or `app.js`: The main application file, responsible for bootstrapping the application.
- `package.json`: Contains metadata about the application, dependencies, and build configurations.
- `App_Resources`: Platform-specific resources (icons, splash screens) for Android and iOS.
- `components`: Directory for reusable UI components.
- `views`: Directory for individual pages or screens of the application.

NativeScript uses XML, CSS, and JavaScript/TypeScript to define the UI and logic of the application.

- **XML:** Defines the UI layout using NativeScript's UI elements.
- **CSS:** Styles the UI elements.
- **JavaScript/TypeScript:** Handles application logic and data binding.

Modules and Plugins

Modules:	NativeScript utilizes modules for extending the core functionality. Modules are typically installed via npm. **Example:** `npm install @nativescript/core`
Plugins:	Plugins provide access to native device features and third-party libraries. They are also installed via npm and often require platform-specific configuration. **Example:** `npm install @nativescript/camera`

Application Lifecycle

NativeScript applications go through a lifecycle similar to other mobile apps. Key events include:

- **launch:** When the application starts.
- **suspend:** When the application is sent to the background.
- **resume:** When the application is brought back to the foreground.
- **exit:** When the application is terminated.

These events can be handled in the `app.ts` or `app.js` file using the `application` module.

Example:

```
import * as application from '@nativescript/core/application';

application.on(application.launchEvent, (args) => {
 console.log('Application launched');
});
```

UI Elements

Layouts

`StackLayout`	Arranges children in a single line, either horizontally or vertically.
`GridLayout`	Arranges children in a grid using rows and columns.
`FlexboxLayout`	Arranges children using flexbox properties, offering flexible and responsive layouts.
`AbsoluteLayout`	Positions children using absolute coordinates.
`DockLayout`	Docks children to the edges of the layout.

Basic UI Components

`Label`	Displays text. Supports basic formatting and styling.
`Button`	A clickable button. Handles tap events.
`TextField`	Allows single-line text input.
`TextView`	Allows multi-line text input.
`Image`	Displays an image from a local file or URL.
`ListView`	Displays a scrollable list of items.

Styling

UI elements are styled using CSS. NativeScript supports a subset of CSS properties, including:

- `color`
- `background-color`
- `font-size`
- `font-family`
- `margin`
- `padding`
- `border-width`
- `border-color`

CSS can be applied inline, in a separate CSS file, or using platform-specific CSS files (e.g., `app.android.css`, `app.ios.css`).

Data Binding

Basic Data Binding

NativeScript supports data binding, allowing UI elements to be dynamically updated based on data changes. Data binding is typically used with MVVM (Model-View-ViewModel) architecture.

Data binding is defined in the XML using the `{{ }}` syntax.

Example:

```
<Label text="{{ myText }}" />
```

In the code-behind (e.g., TypeScript file), the `myText` property is defined in the ViewModel.

```
import { Observable } from
'@nativescript/core';

class MyViewModel extends Observable {
 constructor() {
 super();
 this.myText = 'Hello, NativeScript!';
 }
}
```

Two-Way Data Binding

Two-way data binding allows changes in the UI to update the underlying data, and vice versa. This is typically used with input elements like `TextField`.

Two-way data binding is defined using the `bind` attribute.

Example:

```
<TextField text="{{ myText, mode=TwoWay
}}" />
```

Changes made in the `TextField` will update the `myText` property in the ViewModel.

Event Binding

Event binding allows UI events (e.g., button tap) to trigger methods in the ViewModel.

Event binding is defined using the `tap` attribute (or other relevant event).

Example:

```
<Button text="Tap Me" tap="{{ onTap }}"
/>
```

In the ViewModel:

```
import { Observable } from
'@nativescript/core';

class MyViewModel extends Observable {
 onTap() {
 console.log('Button tapped!');
 }
}
```

Common Tasks

Navigation

Using `Frame`	Navigation in NativeScript is typically handled using the `Frame` component. The `Frame` is a container that holds the navigation history. You can navigate between pages using `frame.navigate()`.
	`import { Frame } from '@nativescript/core';`
	`Frame.topmost().navigate('path/to/newPage');`

Passing Data	Data can be passed during navigation using the `context` property in the `navigate` options. `Frame.topmost().navigate({ moduleName: 'path/to/newPage', context: { myData: 'Hello' } });` In the destination page, access the data using `page.navigationContext`.

HTTP Requests

Making HTTP requests is done using the `@nativescript/core/http` module.

Example:

```
import * as http from
'@nativescript/core/http';

http.request({
 url: 'https://api.example.com/data',
 method: 'GET'
}).then((response) => {

console.log(response.content.toString())
;
}, (error) => {
 console.error(error);
});
```

Common methods include `GET`, `POST`, `PUT`, and `DELETE`.

Platform-Specific Code

NativeScript allows writing platform-specific code using the `platform` module.

Example:

```
import * as platform from
'@nativescript/core/platform';

if (platform.isAndroid) {
 console.log('Running on Android');
} else if (platform.isIOS) {
 console.log('Running on iOS');
}
```

This allows you to use native APIs and features that are specific to each platform.