DART

Dart Web UI Codelab

December 2012

Dart Web UI Codelab

December 2012

Instructor

Shannon -jj Behrens
Developer Advocate
+Shannon Behrens
jjinux.blogspot.com

Introduction

This codelab will help you build a simple chat application using Web Components and the Dart Web UI package in Dart. Along the way, you will learn:

- How to set up pub
- How to use Web Components
- How to use dynamic templates and two-way data binding (inspired by Model-driven Views)
- How to build an application with multiple Web Components
- Where to get more information about Web Components and the Dart Web UI package
- What to do if you get stuck

A quick note about names

We recently renamed the web_components package to web_ui. This new name was chosen to reflect the growing mission for the project. When we first started, we envisioned the project as a polyfill for the new HTML5 Web Components standard. But since then the effort has expanded to encompass a bunch of model-driven views (MDV) and template features, data watching, CSS polyfill work (just starting), and more. This codelab will cover a lot of, but not all of, the new features in the web_ui package.

Prerequisites

This codelab assumes you have already completed [Bullseye - Your first Dart app - Codelab - Google IO 2012](#) (or a more up-to-date version thereof) and that you still have an up-to-date version of Dart Editor installed. Although the code in this codelab is based on the earlier codelab, you'll be starting at a fresh, new starting point since the project layouts are somewhat different.

Additional materials

This codelab provides easy to follow, step-by-step instructions. However, there is a lot of online material that you can use to really master the subject.

Background material

- [Introduction to Web Components](#) is an easy-to-read document from the W3C.
- [The Web Platform's Cutting Edge](#) is a talk the Chrome team gave at Google IO about Web Components.
- [Dartisans Ep. 16: Dart and Web Components Reloaded](#) is an episode of Dartisans that we did with the Dart Web UI team.

Documentation

- [Dart Web Components](#) is the official documentation from the Dart Web UI team.
- [Tools for Dart Web Components](#) is the official documentation from the Dart Web UI team about the tools they have produced.
- Seth Ladd has a bunch of [blog posts](#) on Web Components that I found particularly helpful.
- [Getting Started with Pub](#) is the official documentation for pub, the package manager for Dart.
- [Package Layout Conventions in Pub](#) shows how applications should be laid out in Dart.

Code

- [Web UI](#) is the package I used to build this sample.
- [TodoMVC for Web UI](#) is the sample I referred to the most when writing this codelab.

What to do if you get stuck

See the Troubleshooting section. It's fairly extensive, so don't forget it's there!

Step 1: Import and run the chat app

This codelab walks you through building a custom chat application using Web Components and the Web UI package. You will now load this chat app into the editor and learn how to run both client and server Dart apps.

Objectives

In this section, you will download the source code for the codelab and try out the version in step05.
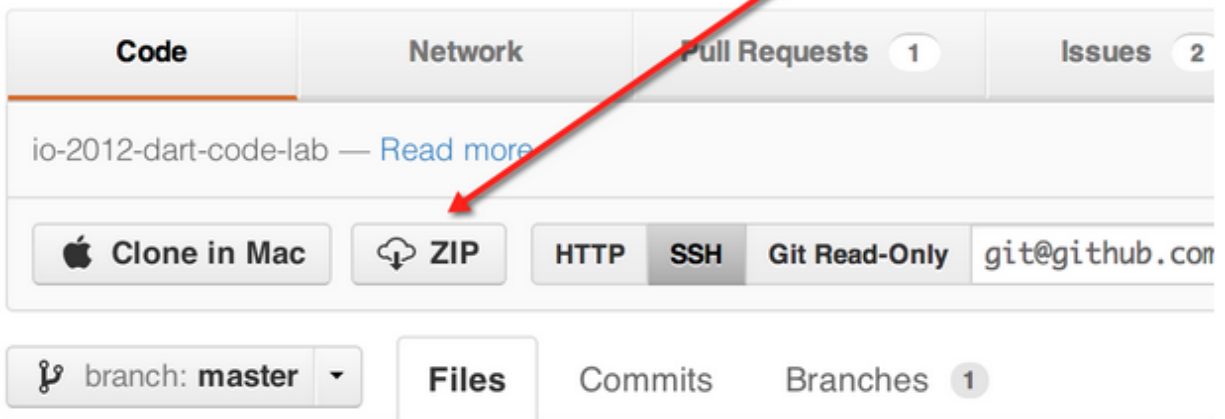
Walkthrough

Download the source code

You can find the source code for this codelab at:

https://github.com/dart-lang/web-ui-code-lab

You can use `git clone` to get a copy of the source code, or you can download a zip of the code by clicking the ZIP button.

If you download a ZIP, be sure to uncompress it.

Load the version in `step05` into Dart Editor

Do not open the entire codelab in Dart Editor. Each subdirectory is its own pub package, so each must be opened separately.

Let's start by opening the version in step05 in Dart Editor. Select `File > Open Folder...` in the editor. Find the `web-ui-code-lab/step05` directory, select it, and click `Open`.

Launch the version of the server in step05

The sample chat app has both a client and a server component.

Run the server first. In the Files view on the left hand side of Dart Editor, navigate into the `step05` directory, and select `bin/chat_server.dart`. Right click `chat_server.dart` and select `Run`.
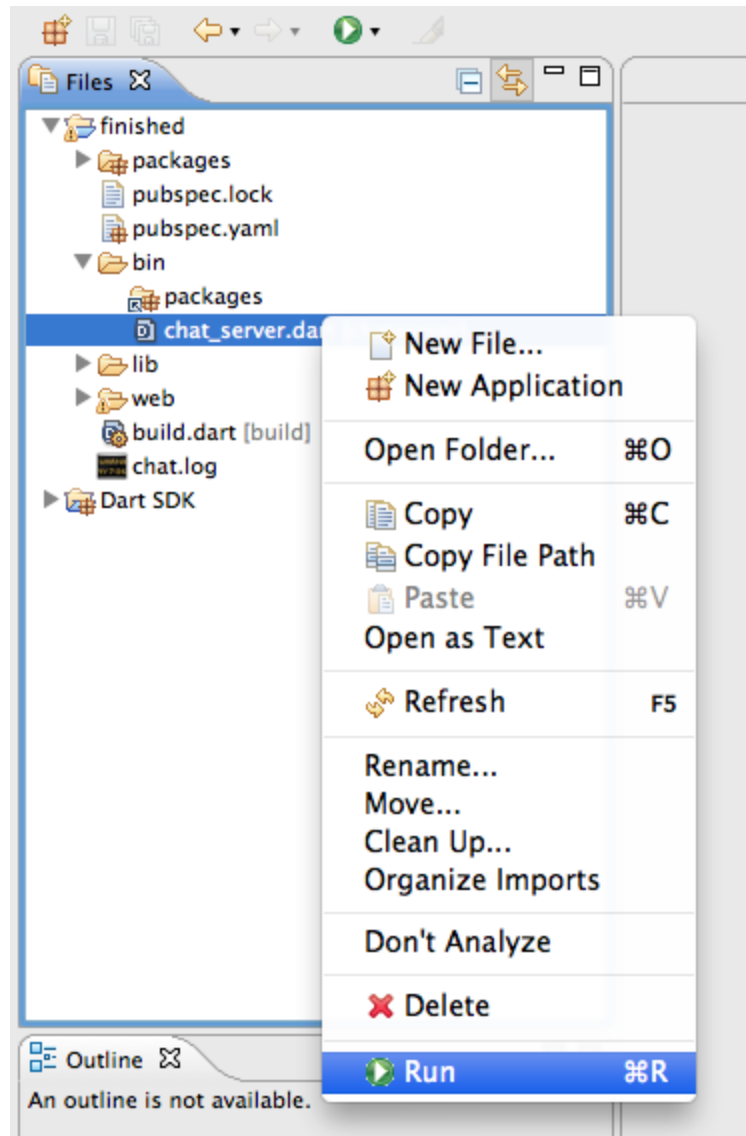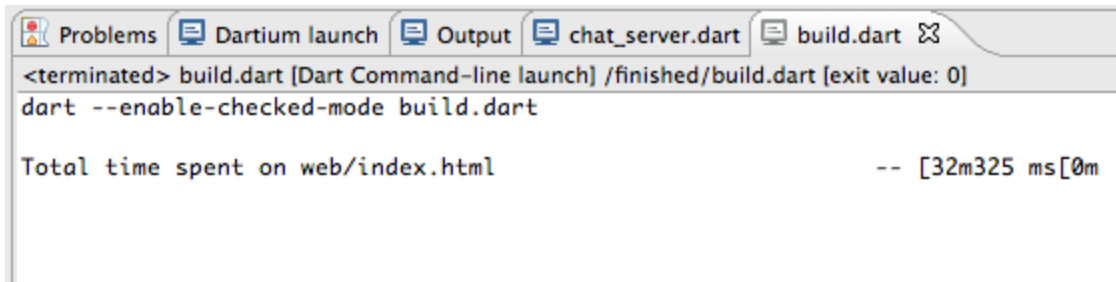
Verify the server is running by checking the `chat_server.dart` console output window at the bottom of your editor. You should see a message: "listening for connections on 1337".

Building and running the client

Although Dart does not need to be compiled to run in Dartium, the Dart Web UI package does use a build step in which the Web Component files are split into various Dart files. Right-click on `build.dart` in the `step05` directory and select `Run`. Look at the output of `build.dart` to check that it completed successfully.

If things completed successfully, there should be a new directory named `web/out` containing the generated output. Right-click on `web/out/index.html` and select `Run`. If everything goes smoothly, Dartium should start, and you should see the client application.
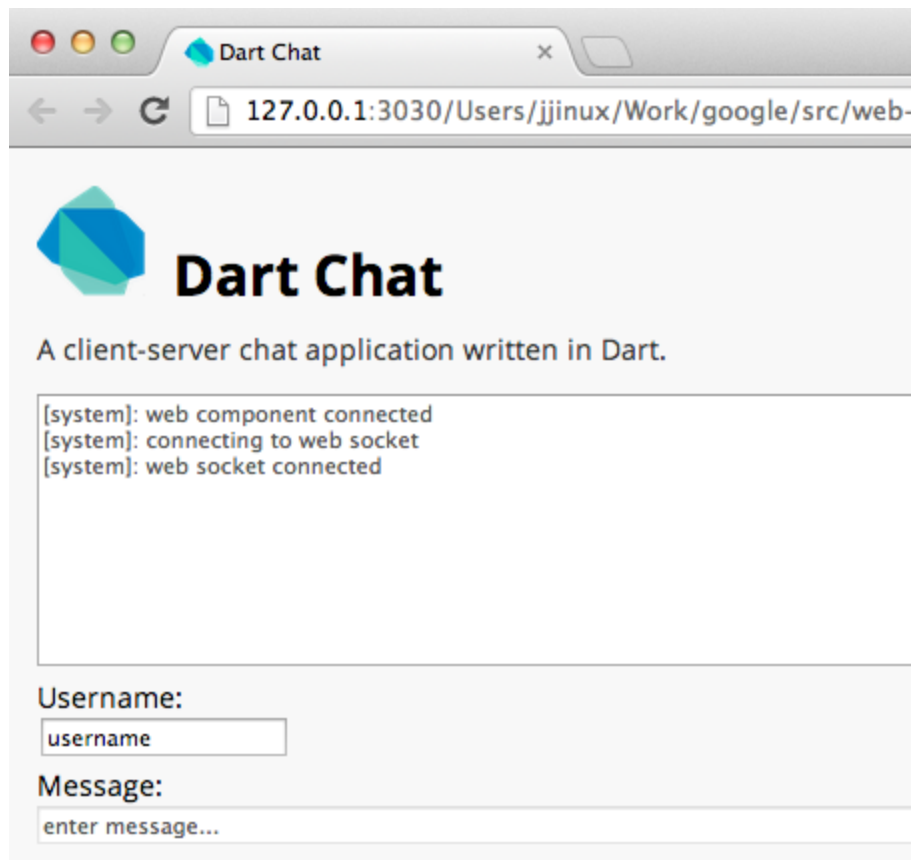
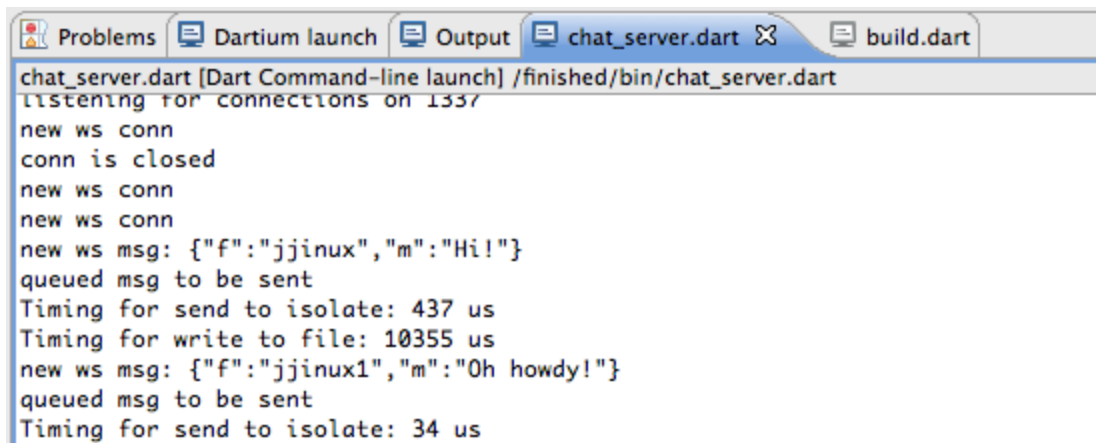At this point, the client should be able to connect to the server. Type in a username and a

message and hit enter. Copy the URL and try it in other browsers such as the normal version of Chrome or Safari.

Debugger view and console output

Switch back to Dart Editor and select the `Tools` > `Debugger` in the top level menu. This lists the two processes that you started, the server and the client.
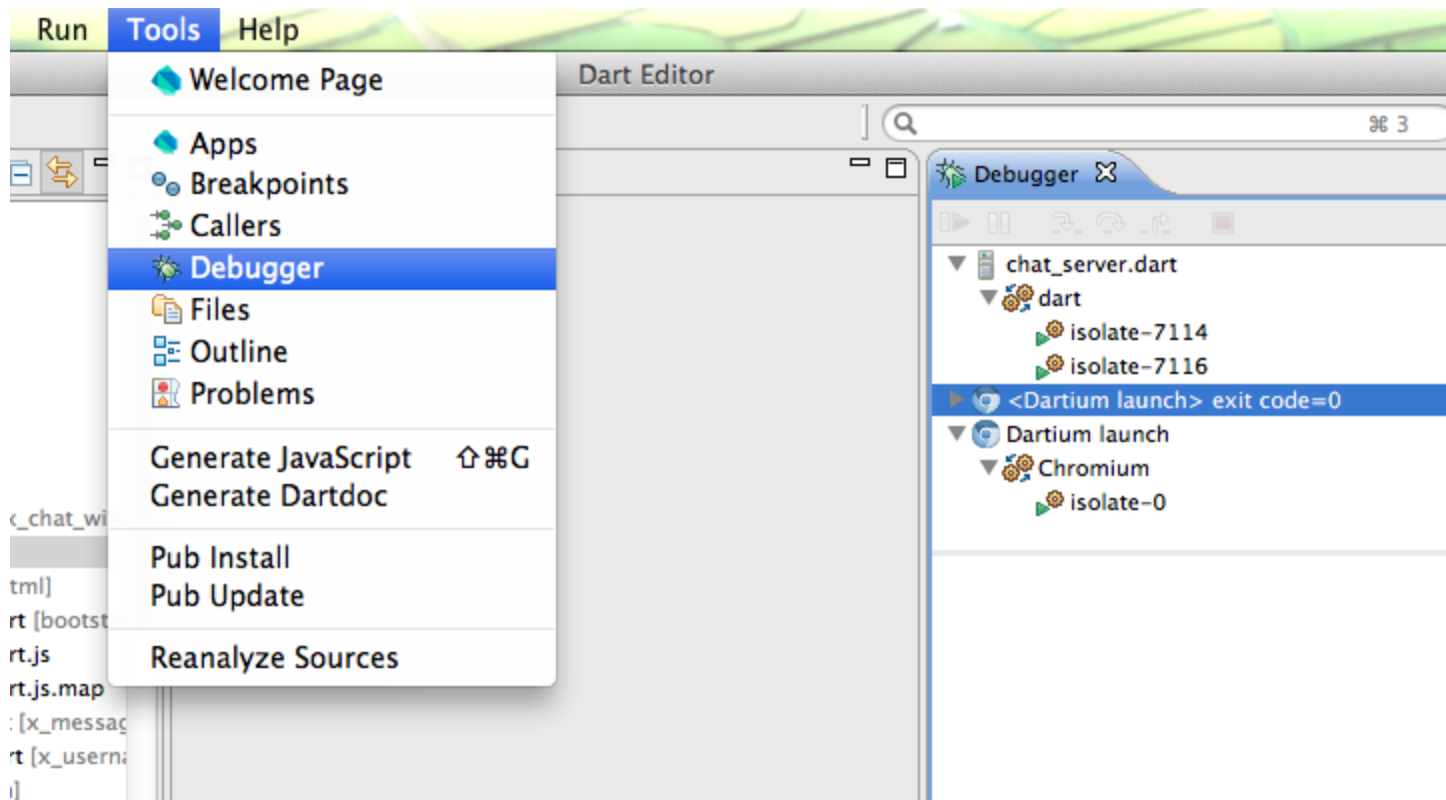
On the bottom of Dart Editor are two views, `chat_server.dart` and `Dartium launch`. Each view has the output from the respective process.

To clear a console's output, click on the gray X icon in upper-right of the console output view. To kill the process, click on the red box in the upper-right of the console output view. After clicking on the red box, you will notice that the Debugger is updated to show that the process was killed.
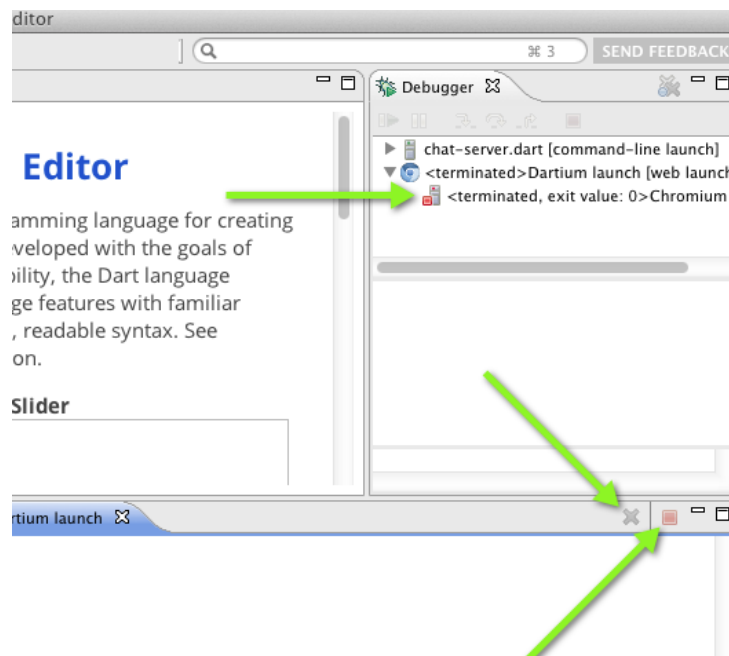
Stop both processes now, first for the `Dartium launch`, and then `chat_server.dart`.

You should also close the `step05` folder in order to remove it from Dart Editor. Right-click on the `step05` folder, and select `Close Folder`.
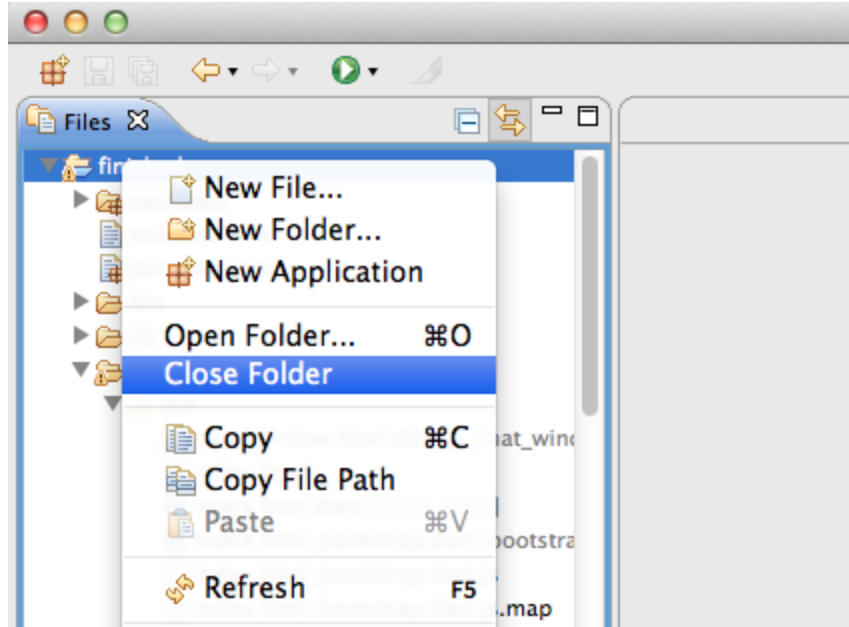
Step 2: Getting started with pub

pub is the package manager for Dart. It is similar to npm in Node.js or RubyGems. A lot of libraries are distributed as pub packages, including the web_ui library.

Objectives

We will start with a stripped-down version of the project in the `start_here` directory and add pub support to it in order to install the web_ui library.

Walkthrough

In the `web-ui-code-lab` directory, make a copy of the start `start_here` directory called `mine`. Open up the `mine` folder in Dart Editor. From now on, you'll be working on that. If you get stuck anywhere, you can either refer to one of the directories, such as step03, or you can overwrite your version of `mine` with a copy of one of those directories if you need a fresh start.

At this point, the project is pretty bare. It has `bin/chat_server.dart` and it's corresponding libraries in `lib`, and there are a few static files in `web`, but that's it.
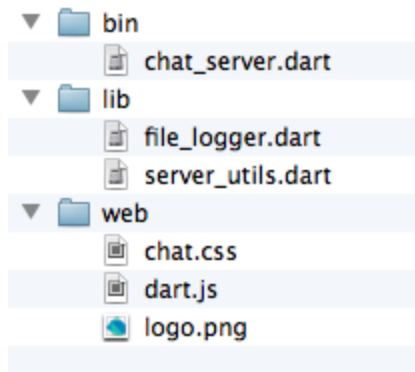
Right-click on the `mine` directory and select New File... Create a new file named `pubspec.yaml`. Put the following in the file:

```
name: dart_chat
description: This is a chat app written in Dart using the Dart
Web UI package

dependencies:
  web_ui: ">=0.2.8 <0.2.9"
```

Notice that we're picking a fairly specific version of the web_ui library to avoid breakages when backwards-incompatible changes are made.

Save the file. Now, run Tools > Pub Install.
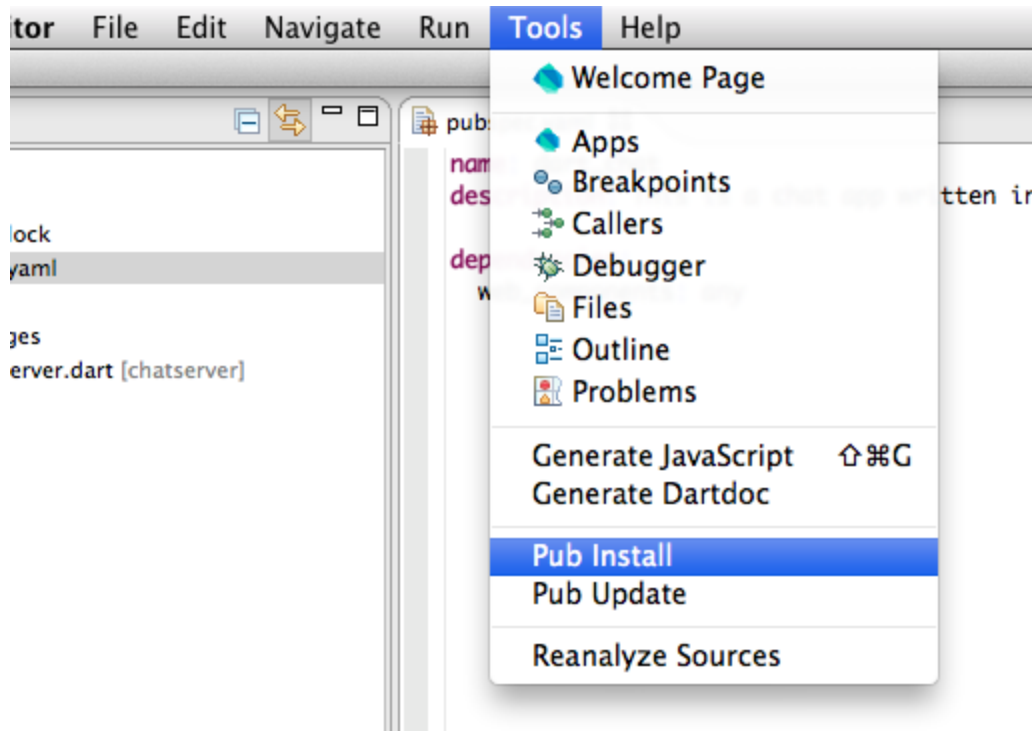
This will result in the following output:
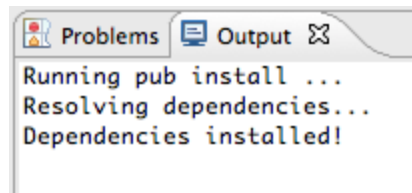
It will also result in some new files and directories:
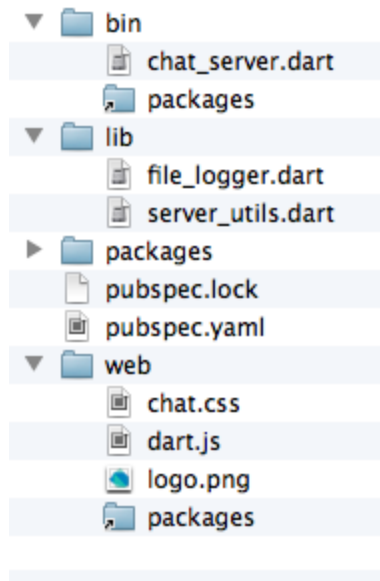
`pub` isn't just used to refer to third-party libraries. It also makes it easier to refer to your own libraries when they are located in the `lib` directory. Open up `bin/chat_server.dart`, and change:

```
import '../lib/file_logger.dart' as log;
import '../lib/server_utils.dart';
```

to:

```
import 'package:dart_chat/file_logger.dart' as log;
import 'package:dart_chat/server_utils.dart';
```

It's convenient to use package-based URLs rather than relative URLs (when doing so is possible) just in case the file you're working on moves to a new directory.

Step 3: Your first Web Component

In Dart, Web Components are stored in `.html` files. They contain both HTML markup as well as Dart code. A build step is used to translate these `.html` files into `.dart` files.

Objectives

In this step, we'll build a basic Web Component. We'll also create a very simple `Application` class and a `build.dart` file that Dart Editor will use to the build the project.

Code

If you need to catch up, you can make a copy of `step02` named `mine` for this portion of the codelab.

Walkthrough

Create build.dart

Right-click on `mine` and select `New File...` Name the file `build.dart`. Put the following in the file:

```dart
import 'package:web_ui/component_build.dart';
import 'dart:io';

void main() {
  build(new Options().arguments, ['web/index.html']);
}
```

Aside from the `['web/index.html']` part, this code is fairly boilerplate.

Create index.html

Now, create a file named `web/index.html` with the following:

```html
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <title>Dart Chat</title>
    <link rel="stylesheet" href="chat.css">
    <link rel="components" href="chat_window.html">
  </head>
  <body>
    <h1><img src="../logo.png" alt="Dart">Dart Chat</h1>

    <p>A client-server chat application written in Dart.</p>

    <x-chat-window id="chat-window"></x-chat-window>

    <script type="application/dart">
      import 'application.dart' as app;
```

```
      main() {
        app.init();
      }
    </script>
    <script type="text/javascript" src="dart.js"></script>
  </body>
</html>
```

There's a lot in this file, so let me point out a few things:

`<link rel="components" href="chat_window.html">` is the code used to link in the
Web Component.

`<x-chat-window id="chat-window"></x-chat-window>` is an example of using a
Web Component. It is a custom element named `x-chat-window`.

`<script type="application/dart">...main() {...}...</script>` is the main
for the application as a whole. You must have a main, even if it's empty. In this code, we're just
calling `app.init()` which we'll create in just a minute.

`<script type="text/javascript" src="dart.js"></script>` works with the
JavaScript output of dart2js so that the application will run in browsers that don't natively support
Dart.

Create application.dart

Now, create web/application.dart with the following:

```
library application;

import 'dart:html';
import 'dart:isolate' show Timer;
import 'package:web_ui/web_ui.dart';
import 'out/chat_window.html.dart';

ChatWindowComponent chatWindow;

init() {
  // The Web Components aren't ready immediately in
  // index.html's main.
  new Timer(0, (timer) {
```

```
    // xtag is how you get to the Dart object.
    chatWindow = query("#chat-window").xtag;
    chatWindow.displayNotice("web component connected");

    dispatch();
  });
}
```

The `application` library is how different Web Components can get a reference to one another. We haven't even built our first Web Component yet, so this is a bit overkill, but it'll come in handy later.

You should be getting an error:

```
Cannot find referenced source: out/chat_window.html.dart.
```

Don't worry, we'll fix that in just a second.

Create chat_window.html

Now, create `web/chat_window.html` with the following contents:

```
<!DOCTYPE html>

<html><body>
  <element name="x-chat-window"
    constructor="ChatWindowComponent" extends="div">
    <template>
      <div>
        <textarea rows="10" class="chat-window"
          disabled>{{chatWindowText}}</textarea>
      </div>
    </template>

    <script type="application/dart">
      import 'package:web_ui/web_ui.dart';

      class ChatWindowComponent extends WebComponent {
        StringBuffer chatWindowText = new StringBuffer();

        displayMessage(String from, String msg) {
          _display("$from: $msg\n");
        }
```

```
        displayNotice(String notice) {
          _display("[system]: $notice\n");
        }

        _display(String str) {
          chatWindowText.add(str);

          // You have to call dispatch whenever Web Component
          // data changes and it's not the result of a user
          // event. That happens a lot for this method.
          dispatch();
        }
      }
    </script>
  </element>
</body></html>
```

That's a lot of code, so let me break it down.

Web Components are HTML documents. The are a mix of HTML and Dart code. Hence, they start with:

```
<!DOCTYPE html>

<html><body>
```

Web Components enable you to create new HTML elements. Here is the code where we create the `x-chat-window` element:

```
<element name="x-chat-window"
    constructor="ChatWindowComponent" extends="div">
```

Notice that the element uses the Dart constructor `ChatWindowComponent`. Also note that all of the examples in this codelab use `extends="div"`.

The next part is the `<template>` for the Web Component. Inside the Web Component is:

```
<textarea rows="10" class="chat-window"
  disabled>{{chatWindowText}}</textarea>
```

Using MDV (Model Data Views), the `<textarea>` will automatically stay in sync with updates to `chatWindowText`. That makes keeping your user interface up to date a snap!

Each Web Component has corresponding behavior. In this case, we have a `<script>` tag with some Dart code.

Each Web Component has a class:

```
class ChatWindowComponent extends WebComponent {
```

At this point, that class must always inherit from `WebComponent`.

The `ChatWindowComponent` class has some data:

```
StringBuffer chatWindowText = new StringBuffer();
```

Updates to the `chatWindowText` object will automatically result in updates to the `<textarea>` as mentioned above.

Next, the class has three very normal Dart methods:

```
displayMessage(String from, String msg) {...}
displayNotice(String notice) {...}
_display(String str) {...}
```

The code calls `dispatch()` after any changes to `chatWindowText`. Generally, this isn't required as long as the modification happens as the result of a user event. However, in this application, the `_display()` method also gets called on application startup as well as when a message is received on the Web Socket. Hence, it is necessary to call `dispatch()` explicitly.

Building and running

Right-click on `build.dart` and select `Run`. Check the `build.dart` tab at the bottom of Dart Editor to make sure there weren't any errors. Also check the `Problems` tab to make sure you don't see any warnings.

Now, right-click on `web/out/index.html`, and select `Run in Dartium`. You should see:
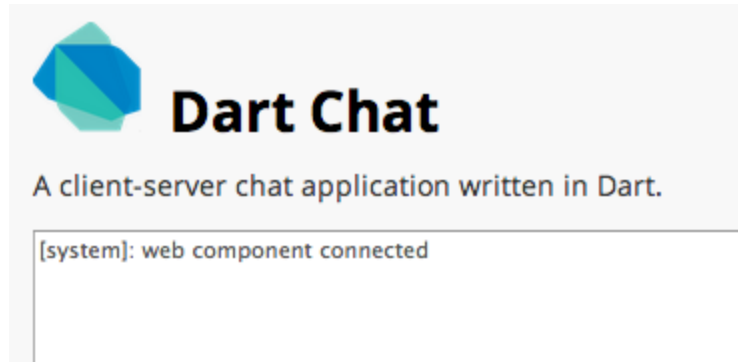
If everything worked correctly, congratulations! You've just built your first Web Component! If you encountered any problems, now's a great time to check out the Troubleshooting section.

Step 4: More, more, more!!!

Objectives

In this step, we'll add two more Web Components, add a Web Sockets chat client, and finish the application.

Code

If you need to catch up, you can make a copy of `step03` named `mine` for this portion of the codelab.

Walkthrough

Update index.html

Edit `web/index.html` (not the one in the `out` directory) and add the lines in bold. These lines correspond to the two new Web Components that we'll be creating.

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8">
    <title>Dart Chat</title>
    <link rel="stylesheet" href="chat.css">
    <link rel="components" href="chat_window.html">
    <link rel="components" href="username_input.html">
    <link rel="components" href="message_input.html">
```

```
    </head>
    <body>
      <h1><img src="../logo.png" alt="Dart">Dart Chat</h1>

      <p>A client-server chat application written in Dart.</p>

      <x-chat-window id="chat-window"></x-chat-window>
      <x-username-input id="username-input"></x-username-input>
      <x-message-input id="message-input"></x-message-input>

      <script type="application/dart">
        import 'application.dart' as app;

        main() {
          app.init();
        }
      </script>
      <script type="text/javascript" src="dart.js"></script>
    </body>
</html>
```

Update application.dart

Now edit `web/application.dart` and add the lines in bold. Once again, we have to add some code for the two Web Components. However, we also have to add some code for the Chat client.

```
library application;

import 'dart:html';
import 'dart:isolate' show Timer;
import 'package:web_ui/web_ui.dart';
import 'chat_connection.dart';
import 'out/chat_window.html.dart';
import 'out/username_input.html.dart';
import 'out/message_input.html.dart';

const connectionUrl = "ws://127.0.0.1:1337/ws";
ChatConnection chatConnection;
ChatWindowComponent chatWindow;
UsernameInputComponent usernameInput;
MessageInputComponent messageInput;
```

```
init() {
  // The Web Components aren't ready immediately in
  // index.html's main.
  new Timer(0, (timer) {

    // xtag is how you get to the Dart object.
    chatWindow = query("#chat-window").xtag;
    usernameInput = query("#username-input").xtag;
    messageInput = query("#message-input").xtag;

    chatWindow.displayNotice("web component connected");
    chatConnection = new ChatConnection(connectionUrl);

    dispatch();
  });
}
```

Create chat_connection.dart

Now create `web/chat_connection.dart` with the following code:

```
library chat_connection;

import 'dart:html';
import 'dart:json';
import 'dart:isolate' show Timer;
import 'application.dart' as app;

class ChatConnection {
  WebSocket webSocket;
  String url;

  ChatConnection(this.url) {
    _init();
  }

  send(String from, String message) {
    var encoded = JSON.stringify({'f': from, 'm': message});
    _sendEncodedMessage(encoded);
  }

  _receivedEncodedMessage(String encodedMessage) {
    Map message = JSON.parse(encodedMessage);
```

```
    if (message['f'] != null) {
      app.chatWindow.displayMessage(message['f'], message['m']);
    }
  }

  _sendEncodedMessage(String encodedMessage) {
    if (webSocket != null && webSocket.readyState ==
        WebSocket.OPEN) {
      webSocket.send(encodedMessage);
    } else {
      print('WebSocket not connected, message '
            '$encodedMessage not sent');
    }
  }

  _init([int retrySeconds = 2]) {
    bool encounteredError = false;
    app.chatWindow.displayNotice("connecting to web socket");
    webSocket = new WebSocket(url);

    scheduleReconnect() {
      app.chatWindow.displayNotice(
          'web socket closed, '
          'retrying in $retrySeconds seconds');
      if (!encounteredError) {
        new Timer(1000 * retrySeconds,
            (timer) => _init(retrySeconds * 2));
      }
      encounteredError = true;
    }

    webSocket.on.open.add((e) =>
        app.chatWindow.displayNotice('web socket connected'));
    webSocket.on.close.add((e) => scheduleReconnect());
    webSocket.on.error.add((e) => scheduleReconnect());

    webSocket.on.message.add((MessageEvent e) {
      print('received message ${e.data}');
      _receivedEncodedMessage(e.data);
    });
  }
}
```

This code is fairly similar to what we used in the first codelab. The only major difference is that the code makes use of `app.chatWindow.displayNotice()` to display messages to the user.

Create username_input.html

Create `web/username_input.html` with the following code:

```
<!DOCTYPE html>

<html><body>
  <element name="x-username-input"
           constructor="UsernameInputComponent" extends="div">
    <template>
      <div>
        <label for="username">Username:</label>
        <input id="username" name="username" type="text"
               bind-value="username"
               on-change="onUsernameChange($event)">
      </div>
    </template>

    <script type="application/dart">
      import 'dart:html';
      import 'package:web_ui/web_ui.dart';
      import 'application.dart' as app;

      class UsernameInputComponent extends WebComponent {
        String username = "username";

        onUsernameChange(Event e) {
          if (!username.isEmpty) {
            app.messageInput.enable();
          } else {
            app.messageInput.disable();
          }
        }
      }
    </script>
  </element>
</body></html>
```

This Web Component is fairly similar to the first Web Component we saw. However, there are a

couple differences.

Look at this line:

```
<input id="username" name="username" type="text"
       bind-value="username"
       on-change="onUsernameChange($event)">
```

`bind-value="username"` means that every time the input is updated, the `username` field (in the `UsernameInputComponent` class) will automatically be updated.

`on-change="onUsernameChange($event)"` means that every time the field is changed, the `onUsernameChange method` (in the `UsernameInputComponent` class) will be called.

Notice the following code:

```
app.messageInput.enable();
```

This is an example of one Web Component talking to another Web Component (by way of the `application` library).

Create message_input.html

Create `web/message_input.html` with the following code:

```
<!DOCTYPE html>

<html><body>
  <element name="x-message-input"
           constructor="MessageInputComponent" extends="div">
    <template>
      <div>
        <label for="message">Message:</label>
        <input id="message" class="chat-message"
               name="message" type="text" disabled
               bind-value="message"
               on-change="sendMessage($event)">
      </div>
    </template>

    <script type="application/dart">
      import 'dart:html';
      import 'package:web_ui/web_ui.dart';
```

```
            import 'application.dart' as app;

            class MessageInputComponent extends WebComponent {
              String message = "enter message...";

              disable() {
                messageElement.disabled = true;
                message = 'Enter username';
              }

              enable() {
                messageElement.disabled = false;
                message = '';
              }

              sendMessage(Event e) {
                app.chatConnection.send(
                    app.usernameInput.username, message);
                app.chatWindow.displayMessage(
                    app.usernameInput.username, message);
                message = '';
              }

              InputElement _messageElement;
              get messageElement {
                if (_messageElement == null) {
                  _messageElement = query("#message");
                }
                return _messageElement;
              }
            }
        </script>
      </element>
</body></html>
```

This Web Component is very similar to the previous one. However, there are a couple interesting parts.

Here's how the Web Component talks to the `chatConnection`, by way of the `application` library:

```
app.chatConnection.send(app.usernameInput.username, message);
```

This code is a getter called `messageElement`. It looks up `#message` and caches it in `_messageElement`:

```
InputElement _messageElement;
get messageElement {
  if (_messageElement == null) {
    _messageElement = query("#message");
  }
  return _messageElement;
}
```

Building and running

First we need to fire up the server. Make sure you don't already have a copy of the server running, and then right-click on `bin/chat_server.dart` and select `Run`.

Since `web/out/index.html` already exists, it's even easier to build and run the client code than last time. Rather than running `build.dart` manually, you can just right-click on `web/out/index.html` and select `Run in Dartium`.

If all goes well, Dartium should appear, and you should see:
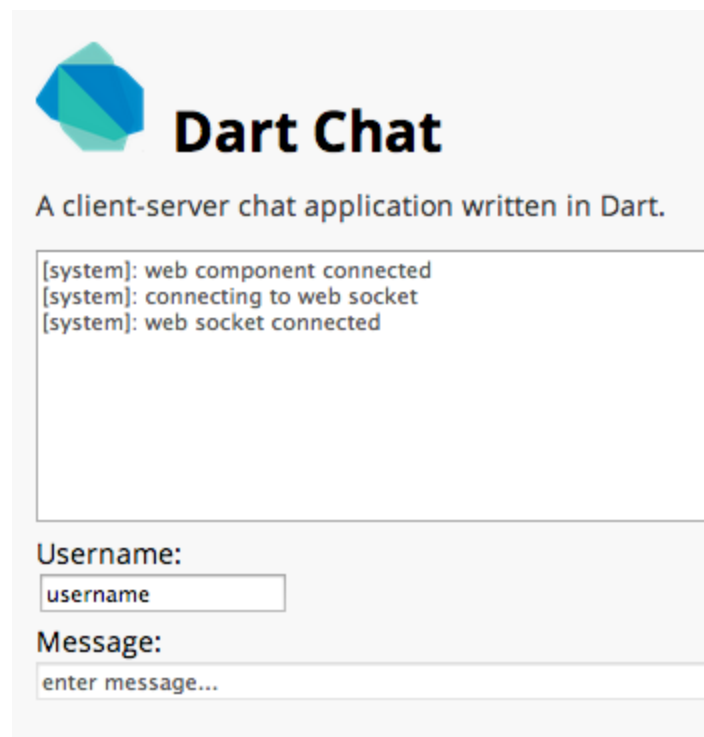
Copy-and-paste the URL into another browser, and see if you can send messages back-and-forth. If everything worked correctly, congratulations! You've just built your first full

application using Web Components! If you encountered any problems, now's a great time to check out the Troubleshooting section.

Step 5: Let's get declarative!

Objectives

In step04, `UsernameInputComponent` and `MessageInputComponent` were coupled in a fairly imperative way. Whenever the username would change, `UsernameInputComponent` would call methods on `MessageInputComponent` to enable or disable the message field. Let's do this in a more declarative way by declaring that the message field should be disabled whenever there is no username.

Also, the user interface felt a little wonky in step 4. The code would submit the message anytime the onChange event fired. There was no Enter button, and pressing return didn't work either. You had to click outside the field or press tab in order to submit your message. Let's clean that up too.

Code

If you need to catch up, you can make a copy of `step04` named `mine` for this portion of the codelab.

Walkthrough

Update username_input.html

Here's the new version of `username_input.html`:

```
<!DOCTYPE html>

<html><body>
  <element name="x-username-input"
           constructor="UsernameInputComponent" extends="div">
    <template>
      <div>
        <label for="username">Username:</label>
        <input id="username" name="username"
               type="text" bind-value="username">
      </div>
    </template>
```

```
        <script type="application/dart">
          import 'package:web_ui/web_ui.dart';

          class UsernameInputComponent extends WebComponent {
            String username = "";
          }
        </script>
      </element>
  </body></html>
```

As you can see, it's much simpler than before! In fact, the only thing that's left is a `username` field which serves as the model for the component.

Update message_input.html

Here's the new version of `message_input.html`:

```
<!DOCTYPE html>

<html><body>
  <element name="x-message-input"
           constructor="MessageInputComponent" extends="div">
    <template>
      <div>
        <form on-submit="sendMessage($event)">
          <label for="message">Message:</label>
          <input id="message" class="chat-message"
                 name="message" type="text"
                 bind-value="message"
                 disabled="{{messageDisabled}}">
          <input type="submit" value="Enter">
        </form>
      </div>
    </template>

    <script type="application/dart">
      import 'dart:html';
      import 'package:web_ui/web_ui.dart';
      import 'application.dart' as app;

      class MessageInputComponent extends WebComponent {
        String message = "";
```

```
            sendMessage(Event e) {
              app.chatConnection.send(
                  app.usernameInput.username, message);
              app.chatWindow.displayMessage(
                  app.usernameInput.username, message);
              message = '';
              e.preventDefault();
            }

            get messageDisabled => (
                app.usernameInput == null ||
                app.usernameInput.username == null ||
                app.usernameInput.username.isEmpty);
          }
        </script>
      </element>
</body></html>
```

It's actually shorter as well. There are a few interesting things to point out.

The `message` input now has `disabled="{{messageDisabled}}"` to control whether or not to disable the input. This refers to a getter called `messageDisabled`:

```
get messageDisabled => (
    app.usernameInput == null ||
    app.usernameInput.username == null ||
    app.usernameInput.username.isEmpty);
```

Rather than imperatively enabling or disabling the field, this code declaratively depends on the model in `usernameInput`.

This code also makes use of a `<form>` with an `on-submit` handler that calls `sendMessage($event)`. There's also an "Enter" button. Now the user can submit messages by hitting the return key or pressing the "Enter" button.

Although, I wouldn't call this user interface the paragon of usability, it's certainly much improved!

Building and running

You can build and run this code in much the same way you did in step04. Try entering and deleting the username field. The message field should be enabled and disabled correspondingly.
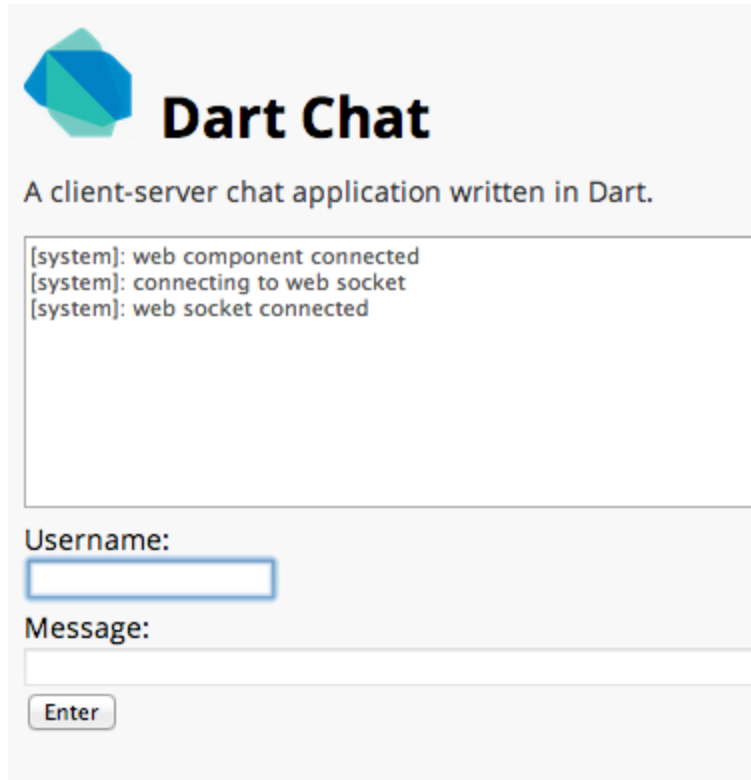
If you encountered any problems, now's a great time to check out the Troubleshooting section.

What's next?

If you made it this far, I hereby pronounce that you are awesome! Here is some Dart code to celebrate your awesomeness!

```
i.did(aWebApp, using: (dartWebUI & webComponents));
assert(i.amAwesome());
```

I suggest you post it to Google+ ;) Just make sure you cc me ([+Shannon Behrens](#)) and tag it #dartlang!

Troubleshooting

The "Dart Web UI" package is fairly new, and it's changing rapidly. This codelab is also very new and has not received a lot of testing. Here are some things to try if you get stuck.

- Unlike the last codelab, each subdirectory of [web-ui-code-lab](#) is a separate pub package. That means you **should not** open up the whole project in Dart Editor. Instead, you should

open up subdirectories one at a time. For instance, you should open `start_here`, `step05`, and `mine` separately.

- The Dart web_components package has [recently been renamed](#) web_ui. Keep that in mind if you see web_components mentioned in any older blog posts, etc. I've done my best to update this codelab.

- If you are working on (for instance) step03 when you get stuck, look at the code in (for instance) step04. step05 is currently the last step, so it's the most likely to be 100% correct.

- If Dart Editor can't find a library that you installed via pub, try running `pub install` on the command line or "Tools / Pub Install" in Dart Editor.

- If pub gets really confused (for instance, if you move your application's directory), try deleting all the directories named `package` and run pub install again.

- If Dart Editor is complaining about something that you have already fixed, make sure all of your files are saved and then use "Tools > Reanalyze Sources".

- If Dart Editor is still complaining about something you have already fixed, it sometimes helps to close the project entirely and open it up again.

- To rebuild all of your Web Components, right-click on build.dart in your project, and select "Run". In OS X, you can just click on the file and hit Command-r.

- The quickest way to test out your code in OS X is to click on `out/index.html` and hit Command-r (to see the shortcut corresponding to your operating system, right click on "out/index.html). In most cases, this will automatically run build.dart.

- **Make sure** you are viewing, editing, and running the right files:
  - Edit the files in the `web` directory, not the `web/out` directory. The ones in the `web/out` directory are autogenerated.
  - Run the files in the `web/out` directory, not the `web` directory. You can't run these files directly until you have used the Web UI tools to compile them.
  - You can view the files in the `web/out` directory to see what your code has been compiled to.

- Remember to start `chat_server.dart`. Otherwise, your chat client won't be able to connect to it.

- You can only run one version of `chat_server.dart` at a time. Make sure you click the red stop button in the debugger to terminate the existing version before starting another.

Make sure you are running the version that you want to be running.
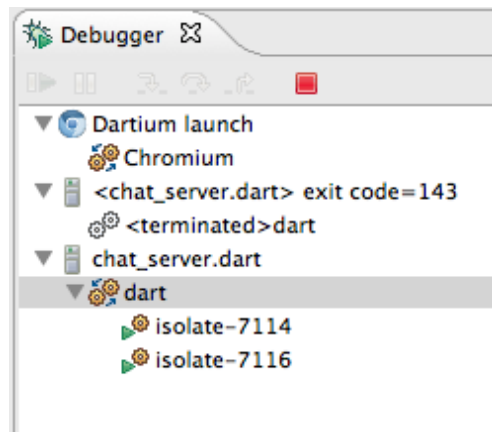
- There are a bunch of tabs at the bottom of Dart Editor. If you encounter problems, check those tabs. The Problems tab will tell you if there are problems in your code. **Warning:** at this point, Dart Editor will tell you if there are problems in the generated Dart code, not in the original Web Component. Hence, you'll need to edit the original file and rebuild. Also keep an eye on the other tabs since they will have output from Dartium, `build.dart`, and `chat_server.dart`.
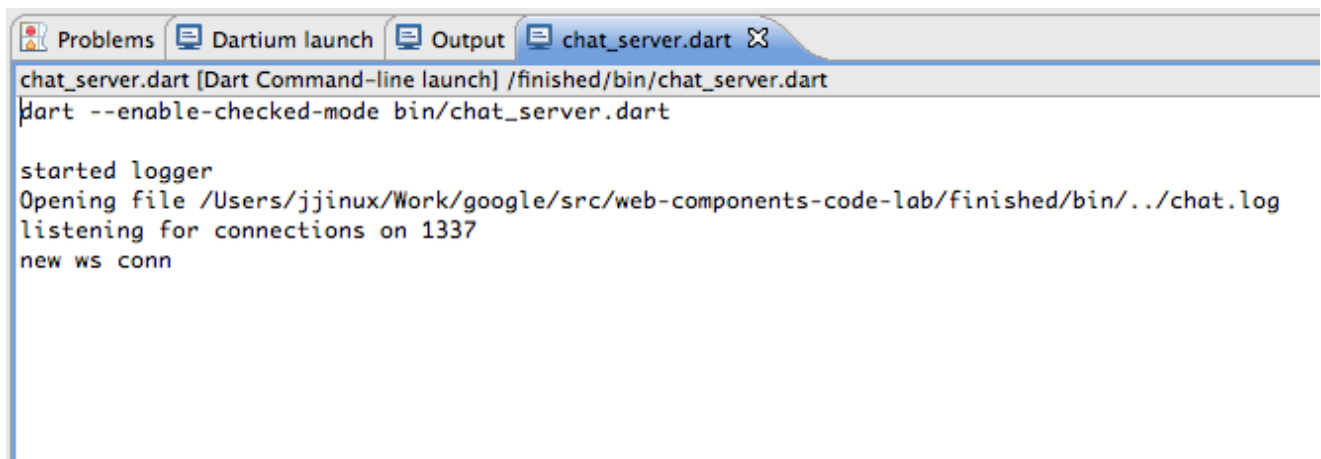
- If you modifying the data for a Web Component, and the user interface doesn't update itself, it could be because you're modifying the data at a time when the web_ui library doesn't expect it (i.e. not as the direct result of a user event). Try adding:

```
import 'package:web_ui/web_ui.dart';
```

And then call the following after you are done modifying the data:

```
dispatch();
```

32

- If you are querying for a Web Component by `id`, and you keep getting `null`, it could be because you are calling `query` in `main` before the Web Component has finished loading. Try adding:

  ```
  import 'dart:isolate' show Timer;
  ```

  And then wrap your code with:

  ```
  // The Web Components aren't ready immediately
  // in index.html's main.
  new Timer(0, (timer) {

    // xtag is how you get to the Dart object.
    var someComponent = query("#some-id").xtag;
     ...
  });
  ```

  This level of indirection will be going away hopefully very soon.

- The syntax for Dart Web Components is not yet set in stone. Make sure the version of dart-web-ui that you are using matches the documentation you're reading. Look at the `pubspec.yaml` in this document to see which version this documentation was written for.

- The dart-web-ui library is always developed with a specific version of the SDK and Dart Editor in mind. You may get warnings if your version of the SDK or Dart Editor are much newer than your version of dart-web-ui or vice versa. The Dart platform is still moving very rapidly!

- Remember that there may be mistakes in the codelab itself. You get bonus points if you can spot them out and submit a bug on dartbug.com!

- If you get stuck while doing this codelab on your own, post your question to Stack Overflow using the "dart" tag.